TURING

Oracle优化日记

一个金牌DBA的故事

白鳝　编著



- 金牌DBA精彩纷呈的经历
- 86个优化技巧活学活用
- 培养多种能力，保障职场成功

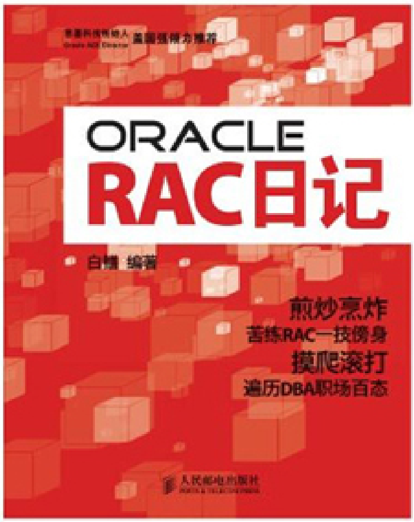
人民邮电出版社
POSTS & TELECOM PRESS

数字版权声明



白鳝　本名徐戟，国内资深Oracle数据库优化专家，从事Oracle数据库应用开发、性能优化工作超过15年。曾供职于DEC、赛格集团、长天集团、联想集团等国内外知名企业，担任过应用体系总监、技术总监等高级技术职务。曾主持开发了国内第一套联机实时计费系统，国内第一套三检合一的检验检疫综合业务系统。1999年起致力于Oracle数据库性能优化等方面的研究，参与了大量性能优化项目，积累了许多实际工作案例。2008年起，在博客上连载《DBA日记》，希望将自己在Oracle数据库应用领域的10多年的经验与广大读者分享。

关于本书和作者的更多信息，可以访问http://www.oraclefans.cn。



图书在版编目（CIP）数据

Oracle优化日记 : 一个金牌DBA的故事 / 白鳝编著.
—北京 : 人民邮电出版社, 2010.7
ISBN 978-7-115-23071-3

Ⅰ. ①O… Ⅱ. ①白… Ⅲ. ①关系数据库—数据库管理系统，Oracle Ⅳ. ①TP311.138

中国版本图书馆CIP数据核字(2010)第096986号

内 容 提 要

本书是一本介绍Oracle数据库优化方法的书，以一个实际的大型优化项目为原型，用日记的形式记录了一个优化小组的DBA如何从纷繁的头绪中找到突破口，进而完成了一个看似不可能完成的任务的历程。在日记之间，作者还穿插了优化小技巧、每日点评等，总结了数据库优化常用的方法、工具和技巧。另外本书第一次详尽地披露了Oracle数据库内部存储结构，并公布了部分代码，对于有兴趣研究数据库内部存储结构或者编写dul工具的读者有一定的参考价值。

本书适合Oracle数据库管理开发人员阅读。

Oracle优化日记：一个金牌DBA的故事

◆ 编　　著　白　鳝
　责任编辑　王军花

◆ 人民邮电出版社出版发行　　北京市崇文区夕照寺街14号
　邮编　100061　　电子函件　315@ptpress.com.cn
　网址　http://www.ptpress.com.cn
　北京　　　　　　印刷

◆ 开本：800×1000　1/16
　印张：22.75
　字数：538千字　　　　　2010年7月第1版
　印数：1－3 500册　　　　2010年7月北京第1次印刷

ISBN 978-7-115-23071-3

定价：55.00元

读者服务热线：(010)51095186　印装质量热线：(010)67129223

反盗版热线：(010)67171154

给初学者的建议

为什么写 DBA 日记

我的 QQ 网名叫“老网虫白鳝”，很多网友都问我为什么起了这么土的一个网名。实际上，这个网名应该是 1998 年的时候起的。10 多年前，我是个十分喜欢玩些新鲜东西的人，ADSL 刚刚在中国出现，我就成为中国第一批使用 ADSL 的宽带用户，时间好像是 1998 年。通过宽带上网确实很快，也拓宽了我在网上的视野，对于用惯了拨号网络的我，第一次感受到了飞一样的上网感觉。有一天中午，和几个朋友一起吃饭，说起了 ADSL，朋友很是羡慕，不过对于我每天只是上网看看新闻，在四通利方上打打嘴仗，感到很是不屑。他建议我去试试 OICQ。在注册 OICQ 账号的时候，我想继续使用在四通利方上的网名“水鱼”，不过“水鱼”很容易让人产生不愉快的联想，于是我准备换一个网名，由于双鱼座的我对水产品的热爱，我想起了中午的午餐“白鳝”。“白鳝”也就是我们常说的河鳗，广东人称之为“白鳝”。结果，QQ 上就多了一条“白鳝”。光阴荏苒，突然有一天我发现我的网龄已经接近 10 年了。于是把网名又改成了“老网虫白鳝”。说实在的，没有“老网虫”，就没有作为 DBA 的“白鳝”。是互联网让我从一名仅仅会安装 Oracle 的入门者成长为一名资深的 DBA。

在网上和网友讨论 Oracle 已经有 10 年了，很多网友问我为什么不写本书，其实我也一直想写本关于 Oracle 的书。2002 年，我产生了把来自 Metalink 的学习成果写出来的想法，书名都起好了，叫《Oracle 深度历险》。书编写了一年多，Word 文档算下来也有 1000 多页。在 2004 年我校对这本书时，发现其中的大多数内容都包含在市面上相关图书中，出版价值并不大。虽然第一次写书很失败，不过写一本书的想法一直没有磨灭。但因工作繁忙，很少有大块空闲时间用以写作，因而也难于写出一本连贯性很强的书来。当年看王强的《圈子圈套》时一下子被迷住了，推荐给很多朋友，看过的人都说在里面能够看到自己的影子，我当时看《圈子圈套》的感觉也是如此。说起来和王强还有过一面之缘，根本就没把他和作家联系起来，但是他的作品在 IT 圈子里的人看来，比作家写得更深入。因为这是他在 IT 圈子里摸爬滚打的经验总结。看过《圈子圈套》后，我开始写《IT AND I》。王强是从一个系统集成行业的高层人物的角度去看问题，而《IT AND I》里的莫明是这个圈子里处于底端的工程师。现在《IT AND I》在我的另外一个博客（blog.oracle-fans.cn/baishan2）里连载，不过最近也已经很长时间没有更新了。我是一个比较懒散的人，也不想给自己太大的压力，只是想把我这些年里做 DBA 的一些经验写出来，共享给大家，所以我决

定写这本书。开始写的时候，我的初衷还是自娱自乐，并没有出书的打算。

直到有一天，我和我的一个同学在北京相聚。他以前在 BEA 公司，在 Oracle 收购 BEA 以后，他成为 Oracle 一个售前部门的总监。我说了一些 DBA 圈子里的事情，他也介绍了 Java 圈子里的一些事情。Oracle 圈子和 Java 圈子全然不同，Java 圈子是一个十分开放的圈子，由于 Java 的开源性，整个 Java 圈子都十分开放，大家都以把自己的工作成果开放出来给大家分享为荣，所以 Java 技术发展十分迅速，技术方面的创新层出不穷。我想，Oracle 圈子只有开放了，才可能像 Java 圈子那样欣欣向荣。从那次开始，把本书公开发表的想法就逐渐形成了。这也是我这一次重新修订《Oracle 优化日记》的一个主要目的。既然目的是正式出版，那么书中的很多内容就不能太随意了，很多观点也要再三推敲，免得误人子弟。

现在大多数的 Oracle 图书都是以技术为主，而且品种十分丰富，但是对于 DBA 来说，要学的不仅仅是技术，还有很多东西不是仅仅通过技术传授所能学到的。作为一名优秀的 DBA，需要具备一些气质、一些性格以及一些处事的方法和原则。这些都不是纯技术的问题，但是往往和我们的 DBA 生涯关系重大。我写这本书的初衷也是为了把我和其他朋友 DBA 生涯中的一些故事介绍给正在学习或者使用 Oracle 的 DBA 们看。本书不是一部小说，因为里面将会介绍很多 DBA 的知识和技术。但是本书也不是一本纯粹的技术书，因为其中穿插了很多故事情节，除了那些风花雪月的事情，其他情感几乎都会在日记里体现。DBA 从事的是一种职业，在职业生涯里也会有喜怒哀乐。除了技术以外，我想我也应该把这些喜怒哀乐传递给大家。

为了叙事方便，我会把发生在很多人身上的事情集中在“我”一个人的身上，“我”不仅仅代表了白鳝，更代表了一批奔四的老 DBA。为了避免一些法律问题，部分客户会使用化名或者故事情节会有所演绎。本书总的来说还是一本技术性图书，并不针对某个人或者某个企业，因此请 DBA 或业界朋友不要对号入座。

另外，十分感谢储学荣、熊文华、罗时峰、邓智国等为本书提供的技术资料和精彩案例、毕建波、薛军和吴刚提供的实验数据，也十分感谢彭国秋、段小真、段小成对本文中文法的校对。当然也要感谢老方、老于、老肖、老熊和小齐，我至今还很怀念几个人在一起喝酒谈论 Oracle 的日子，没有数年前我们的那一次合作，就没有今天的这本书。另外，远在澳大利亚的 Ben 和美国的 John 也给了我极大的支持。在我写《Oracle 深度历险》的时候 Ben 也给予了我极大的鼓励，并且在 2002 年很诚恳地告诉我这本书已经落伍了。Ben 希望我写一些更深入的更有价值的东西。我终于完成了对 Ben 的承诺。

谁适合看这本日记

本书最早是在我的博客上连载的。连载期间，有些朋友产生了共鸣，有些朋友说看不太懂，有些朋友说深受启发。那些产生共鸣的人大概也是资深 DBA 了。本书是比较真实的，虽然也有一些艺术夸大的成分，但也都是有现实依据的。

那些看不懂的朋友，还是没有理解我的意思，实际上本书就像流水账一样记录了一些 DBA 日

常的工作。如果你觉得看不懂，可能是你还没有碰到过那种情况，你不需要理解其中的每个技术细节，看不懂的地方完全可以跳过，这并不妨碍你看其他章节。（当然书中对有些问题的处理过程写得过于简单，读者不容易看懂，因此在为出书而修订时，我已经将这些案例的处理过程细化。另外，对于刚刚入门的 DBA 来说，本书从一个优化案例入手，可能确实会感觉有点不容易摸到头脑。不过不要紧，你完全可以把这本书当做一本写得比较通俗的小说来看，跳过那些生涩的技术描述，提前体会一下 DBA 优化项目时会遇到些什么问题，以及如何面对和处理这些问题。只要你理解了 DBA 分析问题的思路与方法，这本书对你来说就值了。

那些感觉深受启发的人，应该是刚刚进入 DBA 行业的新手。这本书中的很多技术都是这些 DBA 目前正在接触和使用的，而他们又往往缺乏接触大型优化项目的机会。前几天有个网友问我，他正准备接手一个优化项目，能不能给他介绍一下优化项目该怎么做。我推荐他到我的博客上去看看本书。也许和你以前看到的 Oracle 书有点不同，不过书就是书，而不是至高无上的“葵花宝典”。如果你认为看过一本书就能成为高手，那你就错了。

一名老 DBA 的成长之路

结缘 Oracle

从 1993 年第一次帮客户安装 Oracle 算起，我和 Oracle 已有了 16 年的亲密接触。说实在的，第一次接触 Oracle，我对它的印象十分差。在此之前，我只接触过一个大型数据库——DEC 公司的 RDB（现在它已归入 Oracle 名下了）。当时，国内使用的最广泛的小型机平台是 DEC 公司的 VAX，操作系统是 20 年前大名鼎鼎的 OpenVMS。20 世纪 80 年代后出生的人耳熟能详的是 Unix 和 Linux。倒退十几年，20 世纪 90 年代初或者更早的计算机操作系统课程中，很多算法都来自 OpenVMS。那时，在国内使用最广泛的 Oracle 版本是 5.1，而且当时大家的版权意识都比较薄弱，不愿意花几十万元购买正版的数据库，所以当时有一种职业很吃香，即能够帮客户破解和安装系统的工程师。

那时候，我是一名在 OpenVMS 上开发应用的软件工程师，由于工作关系，接触了较多的 VAX 系统。也因当时懂 VMS 和 Oracle 的人十分稀缺，因此常有集成商找我利用周末帮助安装系统。我第一次和 Oracle 的接触就是这样开始的。软件已经破解好，当时有一位老师水平很高，居然写出了一个生成 Oracle 许可证文件的程序，花上 2 000 元就可以买到一个和机器码绑定的许可证，安装介质就是那种 20 年前十分流行的正方形磁带。复制安装介质，编译链接，然后创建数据库，以前的 Oracle 安装十分烦琐，连数据文件都要手工创建后添加到表空间里。当我手忙脚乱地忙活了一天，终于替客户成功安装了一套 Oracle 5.1，并拿到 2 000 元钱的时候，心里别提多高兴了，因为那时候我的月工资不过 1 000 元。虽然钱挣得不算困难，但是 Oracle 给我留下的恶劣印象使我很长时间不愿意接触它。和 RDB 比起来，Oracle 简直太烦琐了，其性能和功能也无法和 RDB 相比。基于这个认识，我在 1994 年帮助泉州电信开发计费系统时，还是全力推荐客户使用 RDB。这个计费系统十分成功，获得了省级的科技进步三等奖。

在这段时间里，我在每个项目中都会碰到大型数据库，要么选择 Oracle，要么选择 RDB，但我更愿意选择 RDB。这期间，Oracle 在不断进步，而 RDB 随着 OpenVMS 在商业上的失败却日薄西山，几年后终于被 Oracle 收购。1995 年，我为一个政府部门设计一套电子单据处理系统时，客户坚持使用开放的 Unix 系统，而拒绝使用 OpenVMS。在 Unix 平台上，Oracle 成了我唯一的选择，当时正是 Oracle 7.1 大行其道的时候。Oracle 7 有了太大的进步，其方便的安装配置以及优异的性能让我感到十分意外。因此，在 1996 年我为泉州电信设计联机实时计费系统时，

Oracle 成了我的首选，正是这个项目让我真正喜欢上了 Oracle。当时使用的服务器是一台有两个 21164 CPU、256MB 内存的 DEC Alpha 2100，虽然今天看来，这台服务器还不如现在的一台普通 PC，但在当时则代表了最先进的 64 位计算的服务器，完成了一个具有上百万市话用户、50 万长话有权用户的大型本地网的联机实时计费系统。在这个项目中，我第一次对 Oracle 进行了优化，通过调优的系统发挥了强大的性能，一个电话在挂机后 5～10 秒就能完成通话话单的结算。后来在这个系统的基础上，福建富士通开发了一个话费回送系统，挂机后几秒钟就可以把通话费回送到客户的来电显示电话上。

从那以后，我和 Oracle 结下了不解之缘。1997 年，我第一次参加了 Oracle Open World，那次北京的盛会除了让我了解了超大内存（VLM）和超大数据库（VLDB），也让我结识了一批 Oracle 第三方服务的先行者——北京巨龙的朋友，他们在 Oracle 这个产业上获得的成功让我羡慕不已。

真正让我成为 DBA 是在 1999 年后。此前，我虽然和 Oracle 形影不离，不过我的主要身份还是一名系统架构师、一名十分优秀的程序员，数据库安装、维护和优化只是我的副业。从 1999 年开始，由于要为一些客户提供专业的第三方技术支持，我开始认真研究 Oracle 的架构及内部原理。我花了差不多两年的时间，在 Metalink 上阅读了至少 2 000 份技术文档，一个专题一个专题地研究 Oracle 的内部原理，从寥寥可数的文字中解密一些 Oracle 秘不可宣的“隐私”，甚至在一些 bug 报告的晦涩文字中寻找 Oracle 内部原理的合理解释。后来由于工作关系，我接触了不少 Oracle 内部文档。我发现若能早点获得这些文档，那么整个学习过程至少可以缩短一半。我觉得 Oracle 公司应该开放这些文档，让愿意深入研究 Oracle 的人员学习。

这段时间，我经常上技术性网站 ITPUB。刚开始的时候，大家的 Oracle 水平都很有限，论坛的学习气氛也十分不错。而随着网站的人气越来越旺，在论坛里大家不再是一起学习和讨论问题了，而是不停地扯皮、争吵，于是 ITPUB 上有一批人转到了 www.oracle.com.cn，我也在这个网站上面混迹了一段时间。DBA 这个圈子保守的气氛使这些网站都很难成为真正高手的园地。在这些 IT 网站上，有价值的内容越来越少，所以我把所有的精力都放到了 Metalink 上。

成长的快乐

经常有 DBA 和我探讨学习过程中的各种迷茫和痛苦。学习 Oracle，在最初的阶段是十分轻松的。Oracle 的技术资料唾手可得，对于初学者来说，这一点十分重要。不过随着学习的深入，很多 DBA 都感觉好像遇到了瓶颈，无论自己多么努力，技术水平想要提高就十分困难了。实际上，DBA 在其成长过程中，所需要学习的不仅仅是技术。现在介绍 Oracle 技术的图书已经相当多了，通过阅读这些图书，DBA 应该能够学到足够的专业知识了。不过大家可能都有这样的感觉，刚开始学 Oracle 时，觉得 OCP 是一道十分高的门槛，总觉得不知道自己需要花上多少时间才能达到那个境界。而事实上，真正想要去考 OCP 认证，花上半年到一年的时间也就足够了。而通过 OCP 认证考试后，自己还是感觉到心里空空的，碰到问题还是找不到解决的方法。

实际上，如果真正认真学习了 OCP 课程，了解了 Oracle 的一些基本原理，并且通过 OCP 考试后，理论上已经具有相当的基础了。只是 Oracle DBA 工作不是考试，OCP 理论也只是一个初

级的理论，如何将这些理论知识融会贯通，实际应用到日常工作中去，是十分关键的，这也就是我们常说的工作方法。作为一名 DBA，除了学习理论知识外，工作方法也是十分关键的。因为 DBA 是一个职业，而不是一门课程，理论知识只是基础，只有理论基础是远远不够的。我碰到过很多正在学习 Oracle 的朋友，他们很容易走入两个极端：一种是仅仅注重理论，看了很多书，但总是感觉看书时好像什么都懂了，一放下书就觉得好像什么都没学到，真正碰到问题时还是两眼一抹黑；另外一种是觉得读书太枯燥，总是喜欢自己摸索，他们哪怕碰到一点点小问题，都会去寻求其他人的帮助，而不愿意自己去书本里学习。实际上，在一个 DBA 成长的过程中，需要将读书和实践有机结合起来，读书固然重要，不过没有实践操作，就无法巩固书中学到的知识。

不过并不是每个人都有条件能够参加各种实践活动的，特别是刚刚入门的 DBA，他们往往缺少大型项目和大型数据库维护的经验，这对于他们在技术上的提高是很不利的。这时其他人的经验就是很好的教材。本书的目的是把我这些年在 DBA 工作中碰到的一些典型案例用一种很轻松的方式说出来，让大家在看这些案例时学习到分析问题的方法。如果看本书时，仅仅去关注那些技术性的东西，那就本末倒置了。本书的真正精华是那些像流水账似的东西，所以大家不要把它当做一本技术宝典来使用，其中涉及的技术都是大家在其他地方能够学习到的。大家更应该看到的是我在碰到各种问题时，是如何处理的，是如何把一些很基本的技术运用到这些项目中去的。记得有一次我去一个客户那里做性能分析，分析结束后，给客户介绍的时候，客户说你说的这些技术我都学过，怎么你能说的如此头头是道，而我却无法把这些知识和实际工作结合起来呢？那位客户不久前刚刚通过了 OCM 考试，按理说他的技术水平和实际操作能力应该是足够的，为什么他无法把知识和实际结合起来呢？实际上这就是工作方法问题，也就是本书想要教给大家的。

每个 DBA 的成长之路是完全不同的，我开始接触 Oracle 时，在国内接触 Oracle 的人还很少。当时唯一能够找到的 Oracle 资料除了 Oracle 随机文档外，就只有太极出的那套 VAX 技术书里屈指可数的几本薄薄小册子了。刚开始接触 Oracle 时仅仅是安装 Oracle，最早是 OpenVMS 平台，后来逐渐转移到 Unix 平台：SCO Unix、DIGITAL Unix、IRIX、SUN OS 等。Oracle 5 和 Oracle 6 在性能优化上没有什么可做的，主要是针对 SQL 进行优化，维护管理也较为简单，主要是表空间管理。那时的系统也比较小，一般都在几百兆字节到几个吉字节之间，所以也很少能够碰到 Oracle 的 bug。不过在这段时间里，有较多的机会接触小型机、网络、操作系统和应用开发，这些经历都让我在今后的 DBA 生涯受益匪浅。随着 DBA 工作做得越来越专一，接触数据库以外的工作就越来越少，到目前为止，可能除了数据库以外，其他技术都基本上停留在理论上了。

对于一名初级 DBA，多接触一下其他技术是十分有好处的。随着时间的推移、年龄的增长，学习能力和学习新东西的速度都会有不小的下降。我曾经在赛格计算机公司工作过几年，该公司是国内最早的系统集成商之一，其创始人查树衡多次出现在徐迟老先生的报告文学里，在 20 世纪 80 年代末是类似于当今杨元庆、郭为这样的风云人物。在赛格计算机工作的几年，有大量机会接触系统基础，接触 VAX 小型机、DEC 网络产品和存储产品，甚至我还帮助公司“攒”了几台小型机。在很长的一段时间里，这段经历对我都有很大的帮助。

特别值得一提的是，应用软件开发方面的经验对我的 DBA 工作帮助良多。DBA 是在与系统和应用打交道，而不是仅仅与数据库打交道，因此应用软件开发、应用软件体系架构方面的经验和知识是必不可少的。在完全成为一名 DBA 之前，我曾经是一名软件工程师、项目经理、系统架构师。我设计过大量的应用软件，因此在分析一个系统时，我往往能够从开发者的角度去考虑问题，在处理问题时比较能够抓住关键，提出的建议也能够切合实际。我经常看到一些 DBA 给客户提出的优化建议，从 Oracle 数据库的理论上来看这些建议没有问题，不过作为一个系统来说，这些建议的针对性不强，可操作性很低，这种建议哪怕提得再多，再深刻，包含的技术含量再高，也是没有多大价值的。

在刚刚进入 DBA 这个行业，特别是刚刚工作时，应该多接触一些应用开发、系统体系架构、IT 架构和硬件方面的知识。这些知识的学习不能停留在表面上，而应该较为深入地去了解。做过系统工程师或软件工程师的 DBA 往往更容易成功。最理想的状态是在做 DBA 之前做过一两年开发，还从事过个把年的硬件工程师工作。实际上，在 DBA 的工作中，不断地要面对应用软件和系统硬件方面的问题。在实际工作中，也会不断地学习这些方面的知识。如果你并没有像我说的那样，在从事 DBA 这个职业之前从事过软件开发或者硬件维护的工作，那也没有什么关系。在 DBA 工作中，尽可能多地去学习这方面的知识就行了。

在 1992 年到 1998 年，我逐渐从安装 Oracle 转向在 Oracle 上做各种应用软件，其中也经常对数据库进行简单的性能分析和优化。在那段时间里，Oracle 相关的图书也逐渐多了起来。通过阅读相关图书，我对 Oracle 的一些基础知识有了一个整体的认识。

在工作几年后，我虽然安装过上百套 Oracle 数据库了，不过对 Oracle 的认识还是停留在很低的层次，于是我想系统地学习一下 Oracle 数据库。在这段时间里，有一本书对我帮助很大，那就是机械工业出版社出版的《Oracle 初学者指南》，这本书不厚，不过对于初学者来说，已经很全面了。后来我也向很多初学者推荐过这本书，但现在在书店里很难找到了。另外，赵松涛编写的《Oracle 9i 中文版基础培训教程》对于初学者入门来说也是十分好的。实际上，Oracle 的官方文档里有一些入门性的文档，对于初学者来说也是很有用的。*2 Day DBA* 就是一本十分不错的入门图书，很适合初学者阅读。

在工作中，我也经常碰到关于性能方面的问题。早期的时候，对于性能的分析仅仅在各种缓冲区的命中率方面，我用 Delphi 编写了一个小程序，把一些常用的脚本编写成一个性能分析小工具，虽然那个小工具在现在的绝大多数 DBA 眼里只能算是一个“小玩意儿”，不过在 20 世纪 90 年代末，到客户那里拿出一个自己写的分析工具，客户还是能够高看你一眼的。随着为更多的客户优化系统，在性能分析方面，我学会了使用 BSTAT/ESTAT 工具，这个工具就是现在著名的 STATSPACK 工具的前身。在 Oracle 7 上，可以使用这个工具来进行 OWI 的分析。不过那段时间里，对于 Oracle 的学习还不是很系统，主要是在工作中遇到什么问题，就去学习什么知识。1999 年，一个偶然的机会我读了一下 *Oracle Concepts*，感觉这本书对我的帮助很大，很多以前工作中碰到的疑点都在这本书里找到了答案。所以我会给每位 Oracle 入门者推荐这本书。认真读几遍这本书，比学习一些独门秘籍要有用得多。*Oracle Concepts* 这本书适合不同层次的 DBA 阅

读，对于初学者，阅读这本书可以打好基础，不过第一次阅读这本书的人，可能很多地方无法理解，其实这时可以跳过那些看不懂的章节。有了一两年工作经验后，再来阅读这本书时，你会感觉这本书能给你带来很多惊喜，很多工作中碰到的匪夷所思的事情，这里都会给你一些合理的解释。这时阅读这本书，还可以让你对你前一段时间的工作和学习做一个梳理，得到一个很大的提升。实际上工作几年后再来阅读这本书的一些章节，还是感觉会有收获。几年前，我粗粗地浏览 *Oracle 10g Concepts* 时，仍然感觉受益匪浅。

在 1999 年，由于我要给几个客户做一些维护工作，所以对 Oracle 的知识做了一些梳理，期间阅读了一些 Oracle 图书，但最大的发现是 Metalink。由于给客户做相关服务，从客户那里拿到了 Metalink 账号，从那时开始，我发现以前想从图书上获取的知识绝大多数都能够在 Metalink 上获取。通过对 Oracle 概念的阅读，我已经基本上掌握了 Oracle 的基本概念和体系，知道了 SGA、PGA 和 UGA 等基本概念，但是这些概念在我的脑子里还是凌乱的、不成体系的、粗浅的。这些概念对于我做一些复杂的分析帮助不大，我需要更加深入地理解这些概念。在 1999 年到 2000 年，由于客户的水平和维护需求也相对较低，所以虽然我在协助客户进行数据库维护，实际上大多数工作都是较为初级的工作。这段时间里我花了大量的精力在 Metalink 上阅读技术文档，这时的主要学习任务是扩大知识面。Oracle 是十分庞大的体系，其广度很大，如果要掌握 Oracle 的一些基本技术，就必须花费足够的时间。在这之前，我的 Oracle 知识主要集中在和应用软件相关的方面。通过这段时间知识面的扩展，我对一些 Oracle 的主流技术、工具基本上都有了一个初步的认识。这段时间的学习对于从事系统架构设计的我来说，帮助也是十分大的，因为这段时间里我面对的主要系统还是使用 Oracle。由于对 Oracle 数据库了解得深入，在我进行系统设计时，就不自觉地从 Oracle 数据库的角度去考虑应用软件，使应用软件能够更加适合 Oracle，更多地利用 Oracle 的新技术。

突破瓶颈

从 2000 年开始，我突然发现我遇到了瓶颈，无论如何看书学习，自己的技术水平就像停滞一样。随便拿一本书翻翻，我都会觉得书上的知识我都看过，好像没有什么东西能够让我提升能力了。这一点让我十分困惑，因为我知道，对于 Oracle 来说，我刚刚入门。有一天，我和海外的一位朋友在 MSN 上聊天，说起了这个问题，他建议我写本书，把我所学到的 Oracle 知识写出来，这可能对初学者有很大帮助。事后我考虑了几天，于是决定写一本关于 Oracle 数据库的书，书名叫《Oracle 深度历险》，这本书包括第 1 章 Oracle 基础知识，第 2 章 SQL 与 Oracle 数据库编程，第 3 章深入了解 Oracle 数据库，第 4 章 OEM 与其他 Oracle 第三方工具，第 5 章备份恢复与容灾，第 6 章 Oracle 数据库性能优化。为了编写这本书，从 2000 年到 2003 年，我阅读了大量 Oracle 数据库方面的技术资料和图书。由于第 3 章的内容是介绍 Oracle 基本原理的，所以我查找了数百篇关于 Oracle 内部原理的技术资料，其中大多数来自 Metalink。在收集资料的过程中，我也得到了美国和澳大利亚 Oracle 公司朋友的大力协助，获得了大量仅限 Oracle 公司内部使用的文档。这些文档对于我理解 Oracle 的内部原理帮助十分大，并且我已经陆续将其发布在 Oracle 粉丝网

（http://www.Oraclefans.cn）上了。《Oracle 深度历险》的编写工作历时 3 年，不过在 2004 年我第二次修改这本书时，还是决定放弃这本书，因为那时候 Oracle 的技术书已经相当丰富了，再出版同样一本书没有任何意义。虽然《Oracle 深度历险》夭折了，不过写书的目标让我在两三年时间里对 Oracle 重新梳理了一遍，对 Oracle 的认识也更加深入了。在写书的过程中，对于每个知识点，如果能够进行实际操作的，我必须亲自操作一遍。这段时间虽然没有写出一本好书，不过写书的经历对我来说是一笔十分宝贵的财富。这几年里，我突然发现 2000 年时我感到的瓶颈在不知不觉中被我突破了。在这之前，我学习了大量零散的知识，虽然利用这些知识可以解决日常碰到的大多数问题，但是我并没有考虑把这些知识串起来，并没有深入地去了解 Oracle 的内部机理，因此我思考问题时往往只局限于某一个知识点。通过这 3 年的写书经历，我习惯于从 Oracle 的内部机理来思考问题，而不仅仅是从某个问题的表象入手。这样我在理解一些 Oracle 原理时，可以更加深入、更加周到。写书带给我的思考使我突破了瓶颈，进入了一个新的境界。

我也经常建议公司的员工，不要光看书，看书时一定要自己亲自做一遍，然后再把做的过程写成文档，书上的东西才能真正变成自己的。其实这个过程包含了几个步骤，第一步是通过读书学到了新的知识，然后通过自己的亲自实践将书中学到的新知识得到一个感性的体验，最后将这个体验用自己的语言描述出来，那么这个知识点就记住了。如果不这么做一下，读书的效果就要打很大的折扣了。

对于 DBA 来说，写博客是一个不错的主意。将自己学习的成果通过博客整理出来，既可以起到整理思路，完善知识点掌握的作用；又可以通过博客和其他 DBA 进行交流，为其他正在学习中的人提供技术资料，最终达到群体学习的目的。我经常建议公司的年轻人群体学习。群体学习说起来很简单，就是如果存在一些知识点，有几个人都想去学习，如果每个人都是独立地去学习可能需要花费一两个月的时间，如果换一种学习方法，就是几个人分分工，每个人学习一个知识点，然后通过互相交流的方式，大家互相传递知识。这种学习方法可能可以缩短一倍的学习时间，而且学习到的知识的深度也会高于一个人自己学习。群体学习的前提条件是，一起学习的人的水平基本接近，而且大家都很开放，都愿意把自己的知识拿出来和大家分享。

大成之道

2000 年的写作使我突破了第一个瓶颈，不过很快我又遇到了另外一个瓶颈，这也是很多 DBA 都会遇到的。在 2003 年到 2004 年，我经常有碰到了天花板的感觉。这段时间里我在经营一家软件公司，最初的时候还自己写一些代码，随着公司越来越大，最后连系统架构都交给了技术总监。从那时候开始，我觉得自己离开发越来越远了，不过在这个阶段我也有更多的时间研究 Oracle 相关的技术。这段时间给客户做优化的项目比较多，在做项目时，总是感觉很难较快地抓住问题的核心。在这段时间里，我阅读了大量 Oracle 内部和优化的相关图书，包括 *Cost Based Oracle Fundamentals*、*Oracle 8i Internal Services for Waits Latches Locks and Memory*、*Inner Look on Oracle Latches*、*Oracle Performance Tuning & Tips*、*Oracle 9i Tuning Guide*（Oracle 官方文档）、*DSI 401E*、*DSI 402E* 等。实际上，每一本书对我理解 Oracle 都有很大帮助，但是哪一本书也没办法解决我

所有的问题，在很多地方涉及内部原理和算法时也往往都是点到为止。DSI 是 Oracle 内部培训教材，也是广大 DBA 追逐的目标。认真学习一下 DSI 对理解 Oracle 内部原理是有很大帮助的。不过想理解 Oracle 内部原理并不是只有 DSI 一条路，说实在的，DSI 的文档，我手头有一些，不过真正认真去看过的也只有 *DSI 401E* 和 *DSI 402E* 这两个。现在 DSI 文档在互联网上很容易下载到。系统地学习一下 DSI 课程对于中级水平的 DBA 来说是个不错的选择。

除了读书外，还有很多问题是书本上无法回答的，所以我也在互联网上查找更多关于 Oracle 内部原理的文档，通过 Google 和百度去搜索更多的 Oracle 技术网站。http://ixora.com.au 是一个相当不错的网站，上面有很多关于 Oracle 内部原理的资料，虽然资料基本上都是基于 Oracle 8 的，但是资料十分权威。ASKTOM 网站也是这段时间我经常访问的网站。最初知道 ASKTOM 是通过 Google 搜索的时候，基本上都有链接指向 ASKTOM，从 ASKTOM 我学到了 Tom 分析问题的思路和方法，这对于我今后自己处理问题是很有帮助的。

至于国内的网站，那时候 ITPUB 和 oracle.com.cn 比较热门，开始的时候也经常在这两个网站上讨论一些技术问题。随着这两个网站上一批 DBA 的水平越来越高，这两个网站上技术交流的质量却越来越低。我也很难在这两个网站上找到所需要的资料，于是把目标转向了英文网站。国外的 Oracle 技术网站很多，不过大多数都是会员制的收费网站，少量免费网站（比如 ITToolbox）上面的技术水平又普遍比较低，所以找了一圈，最后还是觉得学习 Oracle 最好的网站还是 Metalink。由于朋友的帮助，这段时间获得了不少 Oracle 内部的技术资料。通过学习这些资料，我很深入地了解了 Oracle 内部的一些算法。这些学习过程对我突破这个瓶颈起到了至关重要的作用。

在突破这个瓶颈的阶段，我的学习方法是对于某一个知识点，深入地研究下去，并尽可能地把相关知识点融会贯通，特别是两个知识点的结合部。为了了解数据块的结构，我花了相当长的时间去转储数据块，甚至编写了一些小程序去读取数据块中的数据，这时以前搞开发时沉淀下来的深厚的 C 语言功底起到了很好的作用。对于绝大多数 DBA 来说，完成好日常工作并不需要学习那么多内部原理性的东西，不过如果你有兴趣并且有足够的时间，深入研究 Oracle 的内部结构和原理是十分有趣的事情。至少对我来说是这样的，从 bug 报告生涩的文字和少量的内部代码里分析 Oracle 内部实现的原理，和看一部好的小说一样有趣。上大学时和刚毕业时喜欢看一些散文和诗歌，而现在我阅读的对象除了轻松的商业小说外，就只剩下 Metalink 文档了。这是一个爱好问题，可以说已经与 Oracle 技术无关了。

对于绝大多数 DBA 来说，可能他们缺少实践的机会，这对于 DBA 这个职业来说是十分致命的，可能会影响到 DBA 的成长。我第一次做优化项目时，虽然说那时候我已经具备了很深的理论知识，从事 Oracle 维护工作也有几年的时间了，但在项目一开始，还是碰到了很多意想不到的困难，甚至有时候出现了方向上的偏差。这实际上和实际工作经验是息息相关的，哪怕你的理论水平再高，没有真正做过几个优化项目，是很难真正理解什么叫优化的。对于缺乏实际工作经验的 DBA，在 ITPUB、Metalink、OTN 等论坛或者 QQ 群里帮助别人解决问题是一个十分好的办法。你可能会碰到各种各样的案例，它们都是你在目前的工作环境中无法碰到的。通过接触这些实际案例，可以弥补你在实际工作经验上的不足。从 2004 年起，我建立了自己的 Oracle 讨论群

“白鳝的洞穴”，在这个群里，帮助网友解决问题是一件十分有趣的事情。

当你通过了 OCP 甚至 OCM 认证考试后，你的理论知识积累已经达到了一定的程度。这个时候，如果你不能从事相应的工作，在这些工作中把你的理论知识消化并融入你的思维当中，这些理论知识很快就会在你的头脑中消失。一两年后，这些理论知识就会被忘得干干净净。所以说，通过认证考试不是目的，只是一个过程。当然拥有 OCM 证书，会给你的职场生涯带来很多好处。不过认证考试只是一个方面，深入实践是更为重要的。我见过不少拥有 OCM 证书的人，他们有些已经在企业里做管理了，一旦离开 DBA 生产的第一线半年或者一年以后，这些人也和普通的 DBA 没有多少区别了。因此，作为一名 DBA，如果你想一直保持你的技术水平，不要离开生产一线是十分关键的。这当然又是一个悖论了。对于绝大多数人来说，DBA 只是一个职业，一个谋生手段而已，技术只是获得提升的手段，而生活才是永恒的话题。对于我来说，Oracle 是一种爱好，是一种生活方式，所以我最终选择放弃了在大企业做高级技术管理的职位，去从事一个和 Oracle 密切相关的职业。而对于绝大多数人来说，完全没有必要这么执着和痴迷，Oracle 只是生活的一部分，而不是全部。

理论知识的学习是十分枯燥的，很少有人愿意一遍一遍地阅读 *Oracle Concepts*，虽然这么做对你的 Oracle 理论知识的提升帮助很大。和我同龄的人也许还会记得一款很经典的游戏“金庸群侠转”，这个游戏里最牛的武功是什么？不是葵花宝典，也不是独孤剑法，而是主人公刚出生就会的一种拳术“野球拳”，这种拳法的 1～9 级都平淡无奇，不过只要练到 10 级就突然变得无坚不摧了。实际上研究 *Oracle Concepts* 也是一样，如果我们能够通过反复阅读，把 Oracle 的概念融会贯通，一样能够起到“野球拳”的效果。

对于初学者来说，还有一点要注意。那就是 Oracle 的知识面很广，一个人也很难有机会把所有的 Oracle 知识都认真地梳理学习一遍，因此我们在学习时要注意抓住重点，不能胡子眉毛一起抓。在 DBA 学习的初期，尽可能拓宽知识面，了解 Oracle 的方方面面，而一个名资深 DBA 会在某些方面可能有十分深入的研究，但是在大多数知识点上可能只能点到为止，甚至只能简单地了解，直到需要的时候才可能去深入分析。

在这个阶段给别人讲课是学习理论的一个很好的途径，我有一些知识就是通过给别人培训学习的，比如说 Data Guard，对于 Data Guard 的原理，我大体了解，这是从重做日志的结构方面可以推断出来的，但是对具体的一些技术细节以及实现方法就知之甚少了。有一次需要给一个客户讲 Data Guard 的课程，我花了差不多两天的时间来整理培训讲义，并且准备了一个学生实际操作用的实验包，设计了几个常见的演练场景。通过这个备课过程，对 Data Guard 的原理、配置管理和维护的基本操作就了解得十分清楚了。几年过去了，虽然 Data Guard 的技术在不断演进，我在这几年里也没有做过 Data Guard 的项目，但是通过那次讲课，Data Guard 已经深深融入了我的血液里，再也不会忘记了。随着数据库新版本的推出，Data Guard 技术在不断演进，不过其主体技术是不会发生大的变化的，因此在有了一定基础后，只需要你在使用前再去复习一遍相关知识就足够了。

有些知识点可能平时不会注意，但是如果你要给别人讲课，就需要你认认真真地把这些知识

点一个一个搞清楚，因为没有一位老师愿意在课堂上被学生们问得哑口无言。记得刚刚工作时，公司派我给一个客户讲 RDB 数据库，在这之前我连 RDB 数据库是什么都不知道。那时候正在 DEC 公司，老板是香港人，给我留下一本讲义和一本 RDB 的参考手册，给了我 3 天时间准备。那是一项十分艰苦的任务，我给客户讲了近 10 天的 RDB 课程，每天晚上我通过阅读讲义和参考手册备课，对于讲义上的每个实验都要自己先做一遍，第二天再教给学员们。10 天的课程终于讲完了，没有一名学员看出我也是第一次学习 RDB。通过那次讲课，我对 RDB 数据库的了解比公司里其他员工都要深。10 多年后的一天，在一个客户那里听说他们的一个 RDB 数据库里有一批很重要的数据想搞出来，但没有人懂，所以只能通过应用程序显示在终端上，然后由一人一点一点地抄下来。我听说后，凭着那次讲课对 RDB 留下的印象，很快就帮用户把数据导成文本格式，然后通过 SQL Loader 装载到了 Oracle 数据库里。

最后要重申的是，DBA 是一个工作，一个还算不错的工作，但是并不是每个人都需要把 DBA 当做一种生活。因此，对于技术的追求并不是每个人的生活目标。如果你喜欢 Oracle，那你不妨把阅读枯燥的理论知识和干涩的 trace 文件当成一种乐趣，否则就把它当成一种工作，一种生活中的点缀吧。

DBA 的性格

不一定任何人都需要从事 DBA 这个工作，也不是每个人都能成为优秀的 DBA。DBA 是一种压力相对比较大的职业，要求从业人员在工作期间不断地学习新技术。Oracle 数据库每 5 年左右会进行大版本升级，这就需要 DBA 不断地学习新知识。记得几年前在做一个项目时，和一位干了七八年的老 DBA 一起聊天，他说本来想好了，Oracle 9i 的技术就不去学习了，就吃 Oracle 8i 的老本了，不过没办法，想要生存，必须去学习。最后他说他的最大愿望是不要再去学 Oracle 10g 的东西了。不过愿望只是愿望，两年后，我看到他出差时带着一本 Oracle 10g 的书，就说起了那次对话。他也只能笑着说，干 DBA 的都是苦命人，不学习是不可能的。DBA 这个职业可以做得很长，国外一些高手和大师都是从事 DBA 工作超过 20 年的。不过对于绝大多数朋友来说，DBA 只是职业生涯中的一个台阶而已，因此在做职业规划时，首先你需要考虑 DBA 是作为一种过渡性的工作呢，还是作为一种生活和爱好。

要回答这个问题，就需要根据自身的性格来考虑了。有几种性格是不适合做 DBA 的。DBA 需要谨慎的态度，如果你的性格比较急躁，那么 DBA 不是适合你的工作。DBA 承担了企业中最为重要的数据库的维护，其工作性质决定了 DBA 是一种压力十分大的职业。在处理日常工作以及突发性问题时，急躁是最为可怕的性格，越是碰到紧急的问题，越需要 DBA 以冷静的心态来面对，否则很容易出现不必要的问题。2004 年美国的一项调查表明，超过 30%的系统故障是由于维护人员人为失误造成的，因此沉稳的性格是 DBA 减少出现操作失误的一个重要保证。

除了过于急躁外，好奇心太强的人也不适合做 DBA。DBA 在做维护工作时，经常会碰到一些莫名其妙的事情，和自己工作无关的事情尽量不要做，这是铁的纪律。Oracle 公司的工程师到客户现场工作时，一般会拒绝客户提出的和本次任务无关的工作，这也是 Oracle 原厂服务经常被

客户诟病的一点。不过我认为这是一种很职业的态度，我只做和我工作相关的事情。从另外一个角度来说，作为一名负责任的 DBA，从职业的态度来考虑问题，就是做自己技术能力范围内的事情。有些 DBA 无法判断某个操作的风险，在这种情况下，客户让你做某件事情，你到底是做还是不做呢？最好的方式是通过向专家咨询，确认没有问题后再去做。而往往在实际工作中，大多数 DBA 都会尝试干一些自己没有把握的事情。一名好奇心很强的 DBA，可能发现了一个新的脚本，就很急迫地想在自己维护的生产库上尝试一下，可能他根本没有去考虑这个脚本是否存在风险。实际上，在我这 10 多年的 DBA 工作中，也多次出现了由于好奇心强导致做一些自己认为没有风险的事情，结果或多或少地造成了一些问题，甚至有一次我在一个客户的生产库上尝试一个以前没有做过的转储命令，最终碰到了一个 Oracle 的 bug，导致 RAC 的一个节点宕机。从那以后，哪怕再好奇，我也会先充分评估操作的风险，并且尽可能不去做一些和自己工作无关的分析。实际上，作为一名 DBA，是很难经得起诱惑的，因为有很多情况可能你一辈子也碰不上几回。作为一个爱好 Oracle 的人，碰到了某种现场，都可能会被吸引，甚至诱惑。作为 DBA，经得起诱惑，是十分好的性格。从另一方面来说，DBA 需要足够的职业素养，由于 DBA 工作的风险十分高，任何一个违背职业素养的工作习惯都可能演变为工作中的失误，因此做一个真正的职业人是十分关键的。

除了上面讲到的沉稳，DBA 也需要有决断的性格。这似乎有些矛盾，我们虽然强调 DBA 不能胆子太大，但是在某些情况下，DBA 必须决断。有一次客户的数据库出现了严重的问题，导致宕机，启动后没多久再次宕机，客户也十分着急，由于时间十分紧迫，现场工程师和我们在二线做支持的人都没有足够的时间去进行分析，我当时感觉和我以前碰到的一个 bug 十分类似，不过从 Call Stack 来看，还是有些差别。当时现场工程师不敢做这个决定，我说这种时候了，如果这个补丁不起作用，那我们的服务也就做到头了。这种情况下没有别的思路，但是什么都不做肯定是不行的，所以立即打补丁。幸运的是，补丁打上之后，数据库恢复正常了。果断不仅仅是一种性格，这种情况下果断是基于一定的条件的，因为我知道哪怕这个补丁不能解决问题，也是没有副作用的。对风险的理解是果断的基础。所以说这里所说的果断也不是绝对的，不是刚愎自用，不是一意孤行，而是在自己的知识基础上的一种果断态度。因为做出选择不仅仅是依靠勇气，而更多的是依靠自己的底气。

DBA 的责任心是很关键的。一般来说，我面试一名 DBA，首先看到的不是他的技术能力有多强，而是他的工作态度和责任心。一个有责任心的人，哪怕技术水平稍微差一点，也不容易出大问题。而一名缺乏责任心的 DBA，不亚于一颗定时炸弹。能把工作当成自己事情的人是肯定能够成为一名好的 DBA 的。在很多情况下，DBA 的工作都是从纷繁的表象中去发现危险的存在，一个把工作当成苦差事的人是很难做到这一点的。我平时很少和同事发脾气，唯一的一次是因为一件小事，当时客户的一个系统需要我们帮助做一个健康性检查，一共有 10 多套大型数据库，要在两三天内完成巡检工作。当时有 3 个人一起参与巡检，采用的方式是集中采集数据、集中编写报告的方式，这种方式一般来说我们很少采用，因为这种方式可能导致巡检质量下降，但由于时间紧迫，也只能采用这种权宜之计了。在做巡检之前，我就和哥几个说虽然时间紧，但是一定

要认真。虽然哥几个答应得挺好，不过报告出来后，我感觉还是过于粗糙。我只好打回去让他们整改，整改了两三次还是难以让人满意。事后我和哥几个说，如果你把这件事当成一个工作，确实让一个人在这么短的时间里做这么多库的巡检，难免会有些枯燥，质量下降也是难免的，但是如果你是以前的手工艺者，做巡检就是我们的手艺，你拿出的活能不能对得起自己这点手艺呢？大家听后都感触颇深，既然我们吃这碗饭，我们就应该拿出对得起这碗饭的手艺。现代社会比较浮躁，大家都是为了生活而工作，工作已经不是目的而只是手段，这一点我也能够认同。不过人除了物质的东西，总还是需要一些形而上的信仰来支撑自己，否则会失去很多乐趣的。这种信仰就是手艺人赖以生存的基础，失去了这些信仰，把 DBA 工作当成纯粹的谋生手段，那么你还会为了解决一个问题而兴奋不已吗？还会为了自己的失误而感到懊悔吗？

每一个准备做 DBA 这个工作的人，无论自己的职场规划如何，作为 DBA 就应该明白自己承担什么样的责任。摆在我们面前会有很多诱惑，你面对的是企业最为宝贵的财富——数据库。可能你干一辈子的收入还不如把其中一小部分数据复制出去卖给别人赚得多，但是你必须守住自己的信念，你必须对得起自己，对得起自己的衣食父母。记得刚刚工作时，我在 DEC 软件中心帮助香港氧气公司移植他们的核心业务系统，我负责的工作就是将香港氧气公司的 TME 数据库里的数据移植到 OpenVMS 的 RMS 数据库中去。我第一次接触数据之前，老板让我签署了一个保密协议，他当时对我说，“这些数据，随便拿出一些，你就可以卖出几十万的价钱，但是我相信你不会这么做。作为职场中的人，这是最起码的道德底线，今后你可能会遇到很多类似的事情，只要你一次触动了底线，那就万劫不复了。”作为 DBA，那根底线是绝对不能突破的，这不仅仅是道德问题，实际上这个底线是对我们最好的保护。

一个人的性格是天生的，不过也是可以改变的，如果一个人想去做一件事情，并且不断地在努力，成功的机会是很大的。连郭靖这种蠢笨如牛的人都可以成为一代宗师，你想成为一名 DBA 又有何难呢。虽然说不是所有的人都适合做 DBA，不过这一切对于一个努力的人来说，都不成问题。性格是可以改变的，习惯是可以改变的，为了自己的目标，可以改变一切的人，还有什么不能实现吗？我们公司有一个小伙子，性格极为内向，和同事在一起上班，可以一天只说一两句话，甚至一句话也不说。有一次去客户现场工作了两个多月，我们给他一个额外的任务，那就是请客户的 DBA 吃一顿饭。就是这么一个很小的任务，他最后都没有完成。按理说，这种性格的人是很难成为一名合格的 DBA 的，因为 DBA 需要和别人沟通。作为 DBA，三分靠技术，七分靠沟通。就是这样一个内向的人，在大家的努力下，通过一年的时间，居然有了很大的改变，首先是和自己同事之间的沟通多了起来，和客户之间的交流也逐渐好了起来。虽然和其他工程师比较，他还是属于沉默寡言的那一类人，但可以看出，他一直很努力地克服自己的瓶颈，而且我们也看到了他的努力所得到的成果。我想再有一两年的时间，他会成功的。这里我举这个例子就是想说 DBA 的最后一个也是最重要的性格——坚持。大家应该都看过《士兵突击》，许三多不是一个当兵的料，不过他在战友的帮助下，一直坚持着，最后成就了兵王。在这个故事里，有两个重要的要素，一个是许三多的坚持，一个是战友的坚持。钢七连的“不抛弃，不放弃”的信念是成功的关键。对于一个刚刚走入职场想成为一名成功 DBA 的人，这个信念尤为重要。

学习 Oracle 的方法

经常有人问我如何学习 Oracle 数据库，每个人所处的环境不同，所以学习方法也不同。每一名成功的 DBA 可能都有自己成功的道路，但并不是每个人的成功之路都是可以完全借鉴的。学习 Oracle 数据库技术，除了用功外，有很多小技巧是一定要注意的，否则你就会事倍功半。

Oracle 数据库的理论知识相当复杂，而且技术体系也越来越庞杂，对于一个初学者来说，很难抓到学习的要点，往往会觉得自己也很努力，但是成效不佳。在学习的过程中，有两点是要注意的，一是理论结合实践，二是在工作中学习。当你学习 Oracle 数据库时，可能首先会通过看书来学习，看书是学习的最好方法之一，但学习并不仅仅是看书。在看书的过程中，如果你不能把学到的知识实践一下，那么很可能在你刚刚看完一本书时，对这本书的前半部分已经忘得差不多了。如果一边看书一边实践，学习效果就会成倍地提高。如果你能够在工作中学习，那么你的学习效果将会是最好的。因为工作中碰到的问题会让你去思考，并且在思考中才会有更大的收获。前几天和一个以前的同事一起吃饭，他就说："老白，你以前总是让我们把做过的事情认认真真地写下来，当时我们都不以为然，觉得做都做了，写下来有什么用，这段时间我比较有空，就照着你的方法试了一下，发现效果确实很好，写下来的过程实际上是一个知识梳理的过程，在写的过程中也会发现一些知识点存在盲区，这样就会去补充一下，等把这个处理过程完整地写下来时，我发现又学到了不少东西，而且通过这么一写，记得就更牢了。"我们小时候老师经常说一句话"好记性不如烂笔头"，建议我们多写，多记笔记。实际上我们在做一件事时往往更加注重其结果，而会忽略很多过程性的东西，如果做完后，能够系统地写下来，对你的能力的提高是很有帮助的。我经常要求公司里的弟兄们整理知识库，只要他们有空时，我都会发起一个知识整理的活动。知识整理不仅仅是为企业积累核心价值，对于每个参与这项工作的人，都会从中受益。

有一次我正在超市里买东西，公司里的一个小伙子打电话给我，说发现客户有个数据库的 CPU 100%了，从 Top 上看到的都是 Oracle 进程占用了大量的 CPU 资源。那个小伙子是今年刚刚参加工作的毕业生，虽然几个月前通过了 OCP 考试，但临场应变能力以及知识积累方面还存在很多不足，对于日常的维护工作基本能够胜任，碰到一些紧急故障还是感到无从下手。我就问他做了什么分析，他说从 `v$session_wait` 里发现大量的 latch free 等待，由于是 Oracle 8i 的数据库，等待事件并没有明确是什么闩锁。我问他知不知道怎么通过 latch free 的参数找到是哪个闩锁，他说好像以前看到过，记不起来了。我说查看一下 P1、P2 和 P3 参数的说明就知道怎么做了，而且现在需要做一个 STATSPACK 报告，我怀疑这种情况下出现 latch free 等待，很可能是 cache buffers chains 等待。几分钟后，他打电话告诉我，确实是 cache buffers chains 等待，是通过 P2 参数找到的。我告诉他 cache buffers chains 一般来说是由于大量的缓冲区获取引起的，我问他下一步该怎么做？他说："是不是可以杀掉 CPU 消耗比较高的那几个进程？"我说："可以，不过首先要搞清楚是哪些进程，这些进程是前台的还是后台的，是做什么的？"后来他发现有个 SNP1 进程 CPU 开销很大，我问 SNP1 是什么进程，知道吗？他说以前好像学过，现在也记不起来了。我告诉他，是一个和 Job 相关的进程。后来他和客户确认了这个 Job 可以暂停，晚上再跑，就把

这个进程杀了。在杀掉进程后，CPU 还是十分高，又有一批高开销的进程跑了出来，而且这些进程经常会变。他又打电话来问我，我说这时候应该先看看 STATSPACK 报告，关键是注意缓冲区获取排在前几位的 SQL。打完这个电话后很久，他也没有再给我打电话，后来中午吃饭时他给我打了一个电话说这个故障解决了，在 STATSPACK 报告中看到了一个 SQL，在不到 10 分钟的时间里执行了 13 000 多次，消耗了 95%的缓冲区获取，后来客户发现新上的一个模块里面有个循环写错了，写成了死循环，把那个应用停了，问题就解决了。我鼓励了他一番，然后告诉他，希望他根据这个案例，整理一些知识点：一是碰到 CPU 瓶颈时该如何处理；二是 latch free 问题如何分析；三是每个后台进程是干什么的。因为这 3 个知识点是他今天解决这个问题的关键，也正是他暴露出来不足的地方。今后遇到类似的问题，如果这 3 个知识点没有掌握，还是两眼一抹黑。

记得以前联想关于 ISO 9001 有 3 句很著名的话就是："把要做的事情想清楚，把想好的事情做出来，把做过的事情写下来。"这 3 句话在 DBA 身上也同样适用，绝大多数 DBA 都没有整理知识的习惯。很多 DBA 都会感觉自己的知识积累很慢，实际上应该从自身的角度去找找原因，每次实践的机会都是很好的学习机会，而不同的人在每次实践中获得的成果是不同的。善于思考的人会从有限的实践机会中获得更多的知识，主要原因是每次处理完一个案例后你是否能够认真地总结一下。不善于总结的人，哪怕下次碰到完全相同的案例，可能还会像第一次碰到一样手足无措；善于总结的人，下回再碰到类似的问题，都会游刃有余。举一反三这个成语对于 DBA 是十分重要的，只有能够举一反三的人才能更快地提高。

除了实践之外，看书也是 DBA 学习的重要方式，现在市面上 Oracle 方面的图书十分丰富，但是书的质量也是鱼龙混杂，良莠不齐。选择适合自己阅读的图书以及读书的方法也是很关键的。作为一个初学者，并不是任何图书都适合你阅读，循序渐进是非常重要的，过早地去阅读一些自己力不能及的图书是有害无益的。对于一个表空间为何物都不是很清楚的初学者，学习调优或者内部原理是有百害而无一益的事情。实际上，Oracle 官方文档提供了一套很不错的技术资料，从入门级的 *2 Day DBA* 系列开始，到性能调优和一些高级管理技术，都面面俱到。初学者首先自己搭建一个测试环境，然后按照 Oracle 入门书自己一点点实践是一种很好的学习方法。书中看到的一些技术一定要自己亲自操作一下，然后再用做笔记或者写博客的方式，把这些知识点用自己的语言描述出来，这个知识点就算学完了。学完了初级入门的内容后，可以学习一下数据库管理方面的一些基本操作。按照 *Administrator's Guide* 的内容实践一遍所有的操作是一个很好的选择，根据每个人的能力和可以投入时间的多少，这个过程大概需要几个月到一年的时间。

通过这个阶段的学习，基本上对 Oracle 数据库的基本操作有了一个全面的认识，接下来就可以去阅读一本十分经典的图书了，这就是官方文档里著名的 *Oracle Concepts*。由于这本书的理论性特别强，所以阅读起来很枯燥，可能大多数 DBA 都不愿意认真地去阅读它。但是事实上，阅读这本书是很关键的，对于你今后的 DBA 生涯会有较大的帮助。这本书可以帮你建立起 Oracle 基础的理论，有了这本书的理论知识垫底，今后学习 Oracle 的技术将会事半功倍。阅读这本书可能会花费相当长的时间，而且这本书是需要看多遍的，因此第一次阅读时，不一定要求所有的章节都全部认真阅读并理解，有些暂时不使用的技术或者有些对于你来说过于深奥的理论，你完全

可以先跳过去，以后再回过头来阅读。要注意的是，虽然 *Oracle Concepts* 是理论性非常强的图书，实际上每个理论也涉及实际的操作，在阅读时，也要多做试验，以便加深对理论知识的理解。

阅读完这本书以后，需要对我们以前学习的一些知识进行梳理，这时候再去回顾一下 *Administrator's Guide* 中的相关内容是很有帮助的。如果能把实际操作和概念融会贯通，这个阶段的目标就达到了。在回顾的过程中，补充读书笔记，修正以前错误的认识是十分关键的。因为在前面这两个阶段的阅读过程中，肯定会产生很多错误认识，及早纠正错误认识是十分关键的。

上述知识整理阶段结束后，就可以尝试阅读性能优化相关的文档了。实际上，性能优化不仅仅是技术问题，更重要的是理念。学习性能优化相关的技术对于数据库管理和维护是有很大帮助的。学习性能优化，可以从 *2 Day + Performance Tuning Guide* 入手，这份文档里包含了数据库设计、管理、优化的一些最基本的理念，掌握这些理念会对你今后的维护工作帮助良多。官方的 *Performance Tuning Guide* 也是一本优秀的图书，这本书的前几章介绍的内容不仅适用于 Oracle 数据库，甚至适用于所有的数据库，作为应用软件开发和架构设计的参考书也是很不错的。对于初学者来说，学习调优的技术，主要还是强化一些理念，工具和具体的技术细节反而不是最为关键的部分。由于没有实践经验，因此对这本书的阅读不用特别细致，有些地方可以根据自己的喜好跳过，不过这本书的前几章是必须要认真阅读的。

通过上述的阅读过程，一个初学者已经具有了初步的理论知识和一定的实际操作经验，如果你是从事数据库维护的人，就可能会有很多实践的机会，下一步就是如何在实践过程中运用你所学到的知识，以便于将书本上的知识转化为你自己的。如果你不是从事数据库维护工作的，就很难得到实践的机会，你就得想办法去找一份 DBA 工作。在岸上是学不会划船的，要想知道螃蟹的滋味，唯一的办法是亲自吃一只，因此要想成为一名真正的 DBA，你必须首先拥有一份 DBA 的工作。不过换工作不是一朝一夕的事情，或者你可以尝试去考一下 OCP，拿到 OCP 证书，可以为你进入 DBA 行业提供更多的机会。如果你暂时还无法找到 DBA 的工作，那么保持学习状态，并且尽可能地寻找实践机会是十分重要的，否则用不了半年，你以前学习的绝大部分知识都会变得陌生了。实际上，在你等待自己工作转型的过程中，还是有很多知识需要学习。Oracle 是一个庞大的体系，其中的知识点十分丰富，除了数据库的日常管理外，分布式环境、高级复制、RAC、Streams、表分区、EM 等都需要你去掌握，这些都是 DBA 经常要面对的技术。利用这段时间，系统地学习和实践这些知识点，对于你今后正式从事 DBA 工作帮助极大。

在这种情况下，你必须继续学习和实践，通过你自己搭建的测试环境，建立更复杂的测试环境，以便于实践分布式数据库、RAC 等方面的操作。这时加入一些网上技术论坛和 Oracle 专题的 QQ 群是十分有用的。在这些群里和大家一起讨论问题，帮助网友分析和解决问题，是提升自己能力的好方法。

对于有一定基础的 DBA，或者已经从事了几年 DBA 工作的朋友，如何快速提升自己的技术水平是大家都很关心的事情。除了我刚才讲到的积累知识的学习方法外，由于这部分朋友是处在一个爬坡的阶段，因此除了在日常工作中积累工作经验外，还需要对 Oracle 的内部原理进行深入学习。大家都把 Oracle 性能调优看做高水准的工作，也经常在做性能调优时感觉一筹莫展。其主

要原因不外乎两点：一是缺乏开发经验，所以对于 SQL 优化无能为力，而数据库性能优化基本上都和应用相关；二是对于 Oracle 的基本原理认识还不够深刻，无法找到性能优化的关键要点。这些问题都暴露出 DBA 在这个阶段存在的瓶颈。对于这个阶段的 DBA 来说，除了 DBA 日常的维护技术外，还需要了解一些其他方面的技术来拓宽视野，其中十分重要的就是多了解一些开发相关的知识，包括软件架构、开发过程、测试等方面的知识，甚至需要尝试学习 PL/SQL 编程、SQL 编程等方面的知识。除了应用开发方面的知识外，DBA 还需要学习小型机、Unix 操作系统、存储、网络方面的知识，这些方面知识的匮乏是目前大多数 DBA 的通病。Oracle 数据库不是孤立的，其运行环境包括服务器、网络、存储、操作系统等要素，而上面运行的又是一个个特定的应用系统，所以多掌握一些这些方面的技术对于 DBA 来说是很重要的。

除了这些知识外，Oracle 数据库本身的一些基本原理性的东西也是这个阶段需要深入了解的。经过前一个阶段的学习和实践，我们基本上对一些原理性的东西有了初步的认识，不过这种认识还远远不够。这时再阅读一遍 *Oracle Concepts* 对于提升 Oracle 原理的认识有很大帮助，并且会发现很多以前不能理解的问题现在很好理解了，你会感觉这次阅读虽然不是逐字逐句地阅读，但是学到的东西甚至比第一次阅读这本书时还要多。事实上，随着你对 Oracle 认识的加深，每次阅读这本书，都会有新的收获。甚至我现在偶尔翻阅这本书的某些章节，还能从中体会到一些以前忽略掉的东西。在这之前，我们已经掌握了大量的 Oracle 知识，这些知识是以点的形式藏在我们的脑子里的，很多地方还无法串成线，连成面。所以这个阶段是需要把 Oracle 的一些知识点贯穿起来，把相关的知识连贯起来。比如，我们可以从 DB Cache 的基本功能联想到服务器进程如何访问数据块，hash chains 和 LRU 链是怎么工作的，联想到数据库缓冲池和共享池的关系，和 DBWR（数据库写进程）的关系，和 LGWR（日志写进程）的关系，等等。如果你能把这些关系都串联起来，那么今后在分析一个问题时，可以从原理出发，避免走弯路。

在这个阶段，你能够接触到的资料很少，有些原理性的东西只能从网上搜索到只言片语，而且网上的资料里面还充满了错误。在这个阶段，加强 DBA 之间的交流十分重要。如果大家都能把自己手头有限的技术资料共享出来，大家一起来讨论，那么这个过程会容易得多。目前我正在粉丝网上将一些资料陆续发布出来，如果大家对这些资料有兴趣，可以在粉丝网或者其他任何一个 IT 交流论坛上对这些资料进行研讨。如果能够形成一个很好的交流氛围，对大家的进步是帮助很大的。

这些年来经常有人让我推荐几本书，我觉得学习 Oracle 最好的资料是 Oracle 的官方文档以及 Metalink 的“知识库”（knowledge base）。Metalink 应该算是 Oracle 学习者最大的知识库，Oracle 公司也愿意把这个知识库共享给所有用户。在我学习 Oracle 的几年时间里，Metalink 基本上成为我每天必上的网站。每天不到 Metalink 上看几篇技术文档，我就觉得像是缺点什么似的。老实说，在我系统学习 Oracle 的过程中，很少看别人写的 Oracle 图书。除了认真阅读 Oracle 官方文档，我的 Oracle 知识绝大多数来自于 Metalink。

目　　录

5月11日　出　发

今天很匆忙，通过深航确认机票后，就急忙赶到机场，搭乘了 8：50 的 ZH9841 前往沈阳。事实上，对于这次旅行我一点准备也没有，昨天我还在为周末带小孩去海边做着准备。由于这是今年第一次去海边，我特意为两岁的儿子买好了太阳帽、太阳镜、游泳镜和泳裤，还在超市里大肆采购了一批儿子最喜欢吃的食品。结果下午的一个电话就使我不得不放弃海边悠闲生活的计划，开始了这 4 个半小时的长途旅程。

沈阳的优化项目很早就提出来了，不过由于和北方公司的合同问题一直拖着，上星期大家还预计这个项目要在 6 月中下旬才能启动，谁知这几天风云突变，实际开工的时间比原计划整整提前了一个月。虽然合同还在走流程，但是为了配合北方公司整体的时间安排，辽宁的客户和我们的项目经理研究一番后，决定本月 11 日必须启动。

北京的老于和老肖已于昨天晚上登上 54 次列车赶往沈阳，并会参加上午的项目开工会。原本项目经理小齐要求我也一起到场，但由于昨天通知得太晚了，所以我只能今天早上出发，尽快和老于他们会合。那个从去年就开始谈论的优化项目终于要揭开序幕了。我最近这段时间一直比较闲，由于在等待这个项目，也一直没有安排其他事情，所以这回出发得虽然很突然，不过还是有一点冲动的感觉。好久没有遇到这么刺激的全面优化项目了！

昨天下午和老于通过 MSN 聊了个把小时，讨论了一下这个项目的做法。老于在做售前时去过一次沈阳，从系统中采集了部分数据，他的第一印象是，这个项目难度很大，因为主要的系统资源均出现了不足。由于他急着去火车站，所以也没有更多的时间进行交流，最后老于把前几次他采集的一系列数据通过 MSN 发给了我。

深圳飞沈阳是十分漫长的旅行，中间经停南京。在飞机上吃过早餐后，我就打开老于发来的资料，简单浏览了一下，本次优化涉及的系统包括计费、账务和综合客服 3 大系统，几乎涵盖了客户业务的主要系统。从 STATSPACK 报告和 OSWATCH 的信息来看，3 套系统都存在 CPU、I/O 及内存 3 方面的资源问题。CPU 使用率经常在 100%的高位，有几套系统 I/O 也好像有些问题，而且空闲内存只剩下几十兆。从这些数据上看，这又是一个十分棘手的项目，怪不得小齐这回这么大的手笔，老于、老肖、老熊和我都是有 10 多年以上经验的 DBA。昨天我还觉得小齐有点小题大做，什么样的项目需要这么多老鸟一起出动呢？现在看样子是要有一番恶战了。小齐是这个项目的项目经理，是个女孩。老于是项目的现场负责人，具体负责项目的实施。在这个项目里，我的身份是应用优化方面的负责人。

通过 STATSPACK 报告来了解系统的基本情况是我的习惯，长期的优化工作使我对 STATSPACK 产生了一种莫名其妙的依赖症，一个系统如果不通过 STATSPACK 报告分析一下，我总会觉得不太托底。STATSPACK 报告里包含了大量的系统状态信息，只有过一遍 STATSPACK 报告，我才能基本抓住系统的关键要素，同时也能够做出判断，这个系统到底能不能优化，优化后大概能达到什么样的效果。什么事情都是熟能生巧，经过这些年的锻炼，无论是什么系统，只要搭一眼 STATSPACK 报告，对于存在的问题就基本有个概念了。

飞机晚点，在南京停留了很长时间。等待期间，我找了一个有电源的地方，打开电脑继续看老于的资料。直到下午 4 点多了，我才到达沈阳，拖着行李走进靠近北站的邮政大院。沈阳比我想象得热，在深圳出发时穿的短袖衬衫，原来还怕到了沈阳太冷，随身带了一件夹克衫，到了沈阳才发现短袖衬衫居然很合适，夹克衫明显有点多余了。

赶到邮政大院时，老于已经在院子里等我了，高高瘦瘦的，和他电话里的自我描述十分一致。我和老于虽然电话沟通很多，不过见面还是第一次，我们几个也是第一次真正在一起合作。第一次和老于通电话，老于说对我很景仰，我说第一次联络，景仰太不敢当了。老于说听老方说过海尔的事情，而且看过我做云南联通优化项目的文档。云南联通是我和老方一起实施的优化项目，我听老方说后期这个项目移交给了他的一位同事，原来老方说的同事就是老于，看样子我们两个还真有点缘分。

见到老于之前，我一直在担心，把这么多老手集中在一起，每个人都有自己的经验和工作方法，能不能成为一股合力怕是一个问题，我和老熊在一起搭档有一年多了，而和老于、老肖基本上是第一次碰面，如果工作思路和工作方法不一致，很可能导致几个人无法互补，反而互相牵制。这个项目老于是现场负责人，我只负责应用优化部分，而之前的所有优化项目都是我全面负责的，关键的时候都是我拍板下结论，我和老于能否很好合作决定了这个项目的成败。这些顾虑在我和老于第一次会面后就烟消云散了。老于一边说话一边不停吸烟，点上第一根烟后就很少用打火机了，基本上是续着抽。我看到过的抽烟很厉害的人往往都很瘦，看样子抽烟对减肥的功效还是很大的。10 多年的 DBA 生涯，让老于成为一个十分谨慎的人，DBA 行业流行一句话："经验越做越丰富，胆子越做越小。"这句话用在老于身上十分贴切。在来沈阳之前，老方就和我说，老于各方面都相当不错，就是有时候胆子小了点，这方面需要你帮他弥补弥补。对我来说，和一个胆子小的人合作是一件好事，我最怕和胆子大的人合作，DBA 的工作风险十分高，就是胆子很小的人也经常会犯冒进的错误，更不要说胆子大的人了。我说没问题，这种项目，胆子越小，保险系数越高。

第一次和老于见面，我们并没有进办公室，机房门口的楼梯间里放着一张小圆桌和几把椅子，平时作为吸烟区使用。我们两个人就坐在走廊门口的吸烟区聊了起来。以前我抽烟也挺厉害，后来有一段时间咽喉炎很厉害，就彻底戒了，所以我平时基本不抽烟，不过在外面做调优项目时我会抽一些。做项目时稍微抽点烟的好处还是老方告诉我的，老方说有时候在客户现场，总是两个人跑到外面嘀嘀咕咕的给客户印象不好，如果两个人一起出去抽支烟，就很正常了。细想一下也确实有道理，所以后来我每次出去做项目，都会带上几包烟。有时候觉得累了，想出去透透气或

者打几个电话，我都会以吸烟的名义出去一下。

老于给我介绍了一下上午会议的情况。小齐和老方上午也参加了会议，不过会议一结束他们就急忙赶回北京向北方公司的领导汇报项目情况去了。上午的会很简单，会议由客户方的孙主任主持，小齐代表公司做了优化项目的安排。主要是介绍我们的两个文档《XX 项目优化工作进度安排》和《XX 项目分工界面》。小齐介绍完后，客户方的张工代表客户方介绍了目标系统的现状。应用开发商的代表也在会上表态，一定会认真配合优化小组的工作。不过由于开发商的负责人范总有事在云南出差，所以开发商仅有现场支持工程师参会，并无高层参加。会议的气氛很融洽，几方代表都表示会全力以赴来做这个项目。最后老于特别强调，虽然他是这个项目的现场负责人，但他感觉在优化项目方面，我的经验更丰富一些，所以在关键问题的决策上，希望我多拿注意。

聊过之后，我们就来到了办公室。这里是客户的机房，我们就在机房旁边的一间办公室里办公，除了我们之外，只有开发商的一位做应用维护的工程师在这里。客户一般在浑南办公，只有每个月做账务处理时才会到这里来办公。这种工作环境是我所喜欢的，不用在甲方的眼皮底下干活，也就很少会被甲方干扰。曾经有一次做一个优化项目时，和客户的运维部门的办公桌相邻在白天上班的时间，我基本上成了客户的 DBA，不停地帮着甲方做一些琐碎的事情。

老于和老肖简单地给我介绍了一下情况后，我们就一起离开了办公室。半个小时后，我们 3 个人在故宫边上的一个小酒馆里喝了顿小酒。老于是东北人，很豪爽，酒量也不错，老肖一口纯正的北京话，但是酒量还不如我，这种酒老于肯定喝不痛快。我喝酒不行，不过抽烟到还可以，虽然戒烟已经几年了，但因和朋友在一起，抽上几根烟，还是很快活的。

老于感觉这个项目很棘手，系统资源严重不足，应用开销很大，但客户不希望通过扩容来解决问题，而希望通过调优解决所有的问题。在前期的接触中发现应用开发厂商很难配合，这可能成为阻碍项目成功最重大的隐患。老于说，希望我和老熊加入是他看了这个系统后做出的决定，因为他和老肖都觉得这个项目难度很大，而他们两个人对电信的业务不熟悉，做应用优化的把握不大。沈阳这个项目是试点项目，北方公司选择了我们公司和一家做系统集成的公司分别在辽宁和山西做优化项目，所以这个项目的成败又决定了北方九省这个大项目花落谁家。按照老方的话说，这个项目只能成功，不能失败。

从给客户的承诺来看，系统总体性能要提升 30%。要从数字上完成这个目标，不是不可能，但是单纯的数字并不能解决问题，客户的需求是解决目前 3 套系统存在的性能瓶颈和资源不足的问题，减少投诉，而不是玩数字游戏，系统总体性能提升 30%并不能彻底解决目前用户面临的主要问题。这个项目不仅要做到从数字上提升性能，更要彻底解决目前存在的问题，并且保证在大集中之前能够保持稳定运行。从目前老于和老肖掌握的情况分析，除了大规模地优化 SQL 外，还没有看到任何其他的成功途径。

我和老于是风格不同的两种人，对于项目，我向来是乐观的。我虽然很少做出承诺，但是在内心里对任何一个项目都保持乐观。只要是系统，肯定有问题，有问题我肯定能找到——这是我的信条。碰到这种项目，我反而感到有些兴奋，好久没有机会大干一场了。我和老于说，我想办法肯定会有的，这个项目动用了这么强的力量，不可能没有效果的。两瓶啤酒下去后，我就面红

耳赤，心跳加速了。我和老于说，不能喝了，来点饺子就回去吧。老于酒喝得正高兴，谈兴也正浓，不过看到我和老肖喝酒的这种状态，也只好草草收兵了。

回到酒店后，我洗了个澡，翻看了一遍老于刚才拷给我的资料，发现系统中最大的问题是资源不足，一台有两个 CPU 的 P650 居然承担全省销账系统的业务，真让人难以置信，要在这样的系统中进行优化确实难度很大。不管怎么样，一切都明天再说吧。5 个多小时的航程，已经让我疲惫不堪了，现在最需要的是良好的睡眠。

今日点评

老白（"老白"是本书作者的代称。使用第三人称可以排除一些主观的因素，也便于读者理解）从今天开始要去做一个优化项目，希望这次老白的性能优化之旅能够给大家带来一些启示。今天老白在飞机上阅读了老于发来的关于项目的资料。项目没有开始，怎么会有资料呢？一般情况下，优化项目往往在售前阶段就会对用户的系统进行一个初步的诊断，定位客户的系统有什么方面的问题，以便于制订优化目标，因为在和客户签定的合同里，肯定要有优化目标。为了作出合理的承诺，必须在售前阶段采集部分系统的数据，再由专家对这些数据进行分析和评估，从而得到一个合理的目标，并明确地签署在合同中。而这个优化目标将来也会成为验收的重要标准。

在今天，我们还接触到一个概念就是开工会，老外一般称为 kickoff meeting。开工会是优化项目中很重要的一个会议。做优化项目，决不能"悄悄地进村，打枪的不要"，而是要大张旗鼓地进行。在初次入场时，一定要把客户管理这个项目的负责人、接口人等召集到一起，讨论优化计划、接口规则等，同时告知客户所有需要他们提供的东西，还要告知客户其中的一些风险。

比如在沈阳这个项目的开工会上，我们准备了《XX 项目优化工作进度安排》、《XX 项目分工界面》这两个文档。客户向我们提交了《系统优化项目初步安排》、《XX 系统设备及服务现状》等资料。我们向客户申请了所涉及服务器的 Unix 操作系统账号以及 Oracle 权限账号，还有 VPN 临时账号等在本次优化项目中需要的权限。为了方便进出机房，我们还向客户申请了门禁。

从今天的日记里，我们还应该注意到一个问题，就是为什么这个项目需要老白和老熊的加盟，这涉及一个资源调配的问题。需要根据售前对项目的分析做出正确的判断，这个项目中需要什么样的技术人员。如果人员安排不合理，那么在实施过程中将会遇到很多麻烦。

优化小技巧　性能优化项目

Oracle 数据库优化是 Oracle 管理工作中最重要和最复杂的工作之一。数据库优化也是数据库维护的日常工作之一。数据库不是孤立的，而是和一些应用具有紧密关系，数据库往往是应用系统中最关键的部分，因此数据库优化不能完全脱离应用而进行。实践证明，数据库管理维护人员与应用开发者的合作越好，数据库优化就越容易成功。

另外，数据库优化是从应用系统开始规划到应用系统退役整个过程中都在不断进行的，因此，数据库优化是一个长期的工作。很多用户希望通过一次优化就达到一劳永逸的效果，而没有树立

数据库优化工作长期持久的观念，这样往往会导致数据库在长期的系统运行过程中没有得到必要的维护，以至于一旦发生问题，造成很大的损失。

数据库优化可以从多个层次进行：系统级、应用级、硬件资源级等。数据库优化的目的是使数据库的性能满足应用系统的需要，因此在数据库优化时需要制订一个优化目标，经过优化后能够达到预期的目标，优化工作就可以停止。优化的效果和优化的成本是有一定关系的，根据成本选定优化效果是制订优化目标的关键。任何超出最大投入成本的优化都是很难实现的。

数据库优化也是一个相对的工作，大家都知道，系统越大，数据库维护的难度也越大。那么什么是大系统呢？仅仅从数据量来划分大小系统已经过时了。如果有一个系统，其并发量和对数据库服务器的负载无法对数据库服务器（比如说是一台性能十分好的小型机）本身造成很大的负担，那么这个系统的优化工作将十分简单。而对于相同的系统，如果使用一台只有一个 CPU 的微机服务器作为数据库服务器，那么这个系统的负载将很大，对于这个系统的维护将是十分困难的。因此，大系统和小系统不仅仅要看它的数据量，还要看系统的负载与服务器的综合处理能力的对比。平常负载已经达到服务器最大负载 80%以上的系统，可以说是高负载系统，这种系统的维护难度也是比较大的。

数据库优化工作应该成为 DBA 日常工作的一部分，而不应该到了系统出现严重问题后才进行。如果日常优化工作做得好，就不需要经常进行大型的系统优化，系统运行状态也会比较正常，总体维护开销会大幅度下降。

沈阳的客户数据库的数据量并不大，也就是几百吉字节，但由于服务器普遍比较老，配置比较低，所以系统的压力还是十分大的。在这里讲到的 STATSPACK 报告，是做数据库性能分析中经常使用的工具。在 Oracle 10g 及以后的版本中，可以使用 AWR 报告和 ASH 报告来替代 STATSPACK 报告。AWR 和 ASH 报告可以提供比 STATSPACK 报告更为详细的信息。（STATSPACK 报告的默认采样级别为 5，如果做 6 级或者更高的 STATSPACK 采样，会获得更为详细的信息，不过采集快照时的开销更大。一般来说，对于资源比较紧张的系统，不建议做 7 级或者更高的快照。）

从我多年的数据库优化工作的经验来看，数据库优化实际上是在一定资源下，对资源的使用方式进行重组，以达到用户的目标。数据库优化往往要在响应时间和系统吞吐量之间进行权衡。提高响应时间可能需要增加系统的吞吐量，从而加重系统的整体负担。优化系统的吞吐量可能会降低系统的整体负担，但是可能会加大平均响应时间。

从总体来说，数据库优化可以有 3 条途径：

- 调整应用系统，缩短某个事务的响应时间；
- 更加均匀地分配任务，使系统负载降低；
- 调整系统资源的配置。

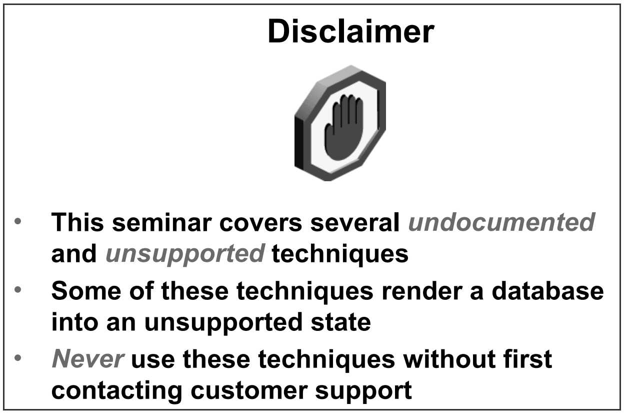
系统资源是有限的，每个计算机系统都有自己最大的负载能力，如果当前整体负载超出计算机的最大负荷能力，这时系统就处于过载状态。系统的任何一个资源一旦过载，数据库性能就会急剧下降。因此在进行数据库优化时，要尽量避免系统的任何资源出现过载，可以通过资源配置

来提高另一种资源的使用，以降低高负载资源的使用。但是如果系统资源过载严重，通过调整资源配置可能也无法达到优化目的，此时增加系统资源将不可避免，这种情况也就是我们常说的系统升级。

优化小技巧　优化常用知识 1：诊断事件

要想做好数据库的性能优化，必须进一步了解 Oracle 的内部结构和一些基本原理。Oracle 诊断事件就是这样的一种知识，它在日常 Oracle 数据库维护、性能优化、故障处理中经常会用到。Oracle 诊断事件是 Oracle DBA 的一项基本技能，必须认真掌握。从今天开始，老白将用几天时间，向大家简单介绍一下 Oracle 诊断事件。这些内容在一些 Oracle 技术书上也有介绍，Oracle 也有一门课程叫做《Oracle 诊断事件》。如果读者阅读过类似的图书，或者已经基本掌握了 Oracle 诊断事件的基础知识，可以跳过本节。对于刚刚学习 Oracle 的初学者，本节的内容可能过于深奥，你也完全可以跳过本节，看完本书后，再回头来阅读。

首先我们引用一下《Oracle 诊断事件》培训课程第一页的内容。



这张 PPT 是一个红色警告，明确告诉每个学习 Oracle 诊断事件的人，Oracle 诊断事件只是一些非官方文档中介绍的非常规知识，并且 Oracle 不会对诊断事件提供技术支持。很多诊断事件的设置可能导致数据库出现不可恢复的故障，甚至导致 Oracle 无法为你的数据库提供技术支持，因此 Oracle 诊断事件在生产库上应该慎用。

一般来说，使用 Oracle 诊断事件的场合有几种，其中最常见的是：当数据库出现了某种故障，要通过设置诊断事件来对数据库进行更深入的分析；其次是通过激活 Oracle 的隐含特性或者改变 Oracle 数据库的运行特性来临时解决某些问题，包括性能问题。由于某个诊断事件在不同的数据库版本上的特性是不同的，因此在设置某个事件时，一定要明确这个事件在你当前使用的版本上的特性，而不能通过其他版本来进行对等的推测，否则可能会导致严重的后果。在使用 Oracle 诊断事件之前，最好找专家或者 Oracle 公司进行咨询，并且在实施风险较大的操作前做好数据库的备份。

Oracle 诊断事件的使用范畴大致可以分为 4 类。

- 在需要的时候转储诊断信息（immediate dump），主要对文件头、系统、进程等信息进行转储。
- 在发生某种错误的时候转储诊断信息（On-Error dump)。
- 修改数据库的运行特性。
- 数据库运行时产生相关日志。

每一个事件都有一个事件号，事件号跟 Oracle 的错误信息的编码是一样的，如 10046 和 ORA-10046。每一个事件都有一个级别，可以是以下几种形式。

- 范围：1 到 10。
- 位标：0x01 、0x02、 0x04 、0x08 、0x10。
- 标识：0=off，1=on。
- ID 号：对象 ID，内存地址。

事件可以通过 ALTER SESSION|SYSTEM 命令来激活和去除激活，语法如下：

```
<event name> <action> {:<event name> <action>}*
<action>::= "trace" <action qualifier>
                  {,<action qualifier>}*
<action qualifier>::= <trace spec>
                    {;<trace spec>}*
<trace spec>::= "name" <trace name>
                <trace qualifier>
              {,<trace qualifier>}*
<trace qualifier>::= ""|"off"|
                     "after <n> times"|
                     "forever"|
                     "lifetime <n>"|
                     "level <n>"
```

事件可以在参数文件中设置，也可以在实例级或者会话级设置。可以通过 ALTER SYSTEM|SESSION 命令直接设置，也可以通过 oradebug 或者 sys.dbms_system.set_ev(…) 存储过程进行设置。

```
参数文件:
  event = "<event name> <action>
         {:<event name> <action>}*"
当前会话:
  ALTER SESSION SET EVENTS
          '<event name> <action>
         {:<event name> <action>}*'
所有的新会话:
  ALTER SYSTEM SET EVENTS
          '<event name> <action>
         {:<event name> <action>}*'
其他会话:
  sys.dbms_system.set_ev
(sid,serial#,<event>,<level>,'<action>')
  In another session from a debug tool:
  oradebug <command>
```

5月12日 开始工作和 CPU 瓶颈

今天上午和客户在邮政大厦的主机房碰了个面。都是东北人，所以大家也没多讲什么客套话，上来就直入主题。张工他们一来，就让我看了一下他们的内部 BBS，里面充满了对这个系统的抱怨与期望。我以前做海尔优化项目时碰到过类似的情况，那时候海尔全国的操作人员组成了多个 QQ 群，在里面发泄自己的不满，老万每天都在群里找出几句最恶毒的话来刺激我的神经，希望我能够更快完成项目。我现在虽然对此有所准备，但还是被那些 BBS 上的留言深深打动了。由于这个系统的问题，甚至出现了愤怒的客户把营业台的玻璃砸碎，冲进员工区殴打营业员的恶性事件。

看完 BBS 后，张工说这回全靠你们了，为了拿到优化这个项目的试点，他们领导差点和兄弟省的 IT 部门打起来。被圈定为优化试点后，他们又选择我们来做实施，没有选择那家著名的集成商。如果这次无法获得满意的结果，那么他们部门的压力将会很大，甚至领导的位子都会出现问题。

我终于能够理解昨天老于的心情了，这样一个项目，参与者的压力和责任是可想而知的。运维部门面对这样的系统，承受了相当大的压力，他们也希望通过这一次优化，能够打一个翻身仗。当这种压力传到老于身上时，作为项目经理，他的忧虑是十分正常的。从这上面我也感受到一点安慰，以我的经验来看，这样的客户肯定会全力配合我们的工作，在和甲方的关系上，不需要我们花太多的精力。

经常有朋友问我，数据库优化应该怎么做。实际上，数据库优化并不是什么高深的学问，和通常解决问题是一个思路，都要经历了解情况、提出方案、实施方案、评估结果这几个步骤。因此，了解系统的情况以便于下一步根据系统实际情况提出合理的优化方案是十分重要的。因此我做系统优化的方法一般是先从摸清系统整体情况开始，包括系统现状、存在的主要问题以及应用系统的特点等。如果可能，还要尽可能地了解清楚硬件配置、存储和应用软件的体系架构等。

我们面对的目标系统是计费、账务、综合客服三大系统。今天上午初步看了一下三套系统的基本情况。综合客服系统，主机配置 4 颗 CPU，CPU 资源在业务进行期间，基本可以满足业务需要，不过时有使用率增大的现象，业务高峰期间经常达到 100%。在平衡运行期间，CPU 平均空闲为 70%，繁忙时段空闲为 5%左右，系统响应速度慢，应用等待时间长。系统共配置 8GB 内存，只分配给了 Oracle 数据库 2GB 左右，但是空闲内存只有 150MB，甚至部分时段只有不到 100MB 空闲内存。存储全部采用 RAID 5 技术划分，读出效率高，可是写入效率不高，对于频繁

需要写入的日志文件和控制文件的操作就存在着瓶颈。业务高峰期事务平均响应时间为 100ms 左右。

计费系统，主机配置 6 颗 CPU，CPU 资源在业务进行期间基本可以满足业务需要，时有使用率增大的现象。在平衡运行期间，平均空闲为 70%，繁忙时段空闲为 15%左右。系统共配置 12GB 内存，其中只分配给了 Oracle 数据库 2GB 左右，实内存剩余 9GB 以上。但是空闲内存只有 500MB 甚至更少，时常出现内存换进换出的现象。存储也是采用 RAID 5，I/O 性能不佳，业务高峰期间 I/O 等待（WIO）高达 50%。业务高峰期事务平均响应时间高达 27 000ms。

账务系统，主机配置 2 颗 CPU，在平衡运行期间平均空闲为 65%，繁忙时段空闲为 0，业务高峰期间，CPU 的运行等待队列高达 100 以上，甚至超过 200，系统响应缓慢，CPU 存在严重的性能瓶颈。系统共配置 8GB 内存，只分配给了 Oracle 数据库 2GB 左右，在业务繁忙期间剩余的实内存只有 100MB 甚至更少，业务期间出现频繁的换进换出现象，内存严重不足。业务高峰期间，I/O 等待（WIO）在 30%～40%，从 STATSPACK 报告上看，系统 I/O 性能不佳。业务高峰期系统事务平均响应时间高达 3 800ms。

面对这样三套系统，我觉得优化的难度还是比较大的。在以往参加的优化项目中，锦上添花的比较多，系统虽然存在一些性能问题，但是对生产的影响还不是十分大，所以优化的目标也比较容易达成。而这种已经基本上算是病入膏肓、资源严重不足的系统，还是比较少见。况且客户的期望不仅仅是说要从技术指标上提升多少倍，还要通过这次优化，彻底解决他们在 IT 运维中无法解决的难题，使运维人员不要每天疲于应付来自四面八方的投诉。

中午和张工他们在旁边的一家小餐馆里一起吃了顿便饭。张工的年龄和我们相仿，以前是乙方的，到甲方来时间还不长，不过一来就遇到这么个烂摊子，也真是难为他了。我说要不要喝点酒，张工立马制止了："中午就算了吧，回去被领导看见不好。我也很想和你们好好喝一顿，不过希望是在优化完成时。听说老于酒量不错，到时候我们喝个痛快。"

下午账务系统和业务支撑系统又出现了大规模的投诉，集成商负责现场维护的小孙跑出来找我们时，我和老于、老肖正在吸烟区吞云吐雾地讨论下一步如何安排。老肖不抽烟，被夹在两根烟枪中间，却也没显得不适应，估计他是被老于这杆烟枪熏陶习惯了。我问小孙怎么回事，小孙说大量营业厅在投诉，说慢得无法操作，他自己也试了一下，登录系统就要一分多钟。

我和老于急忙掐了烟，跑进机房。到现场一看，收费系统的 CPU 等待队列高达 200，随便一个 ps 命令都要两三秒才有反应，看样子系统资源是遇到严重的瓶颈了。这几天是收费的高峰期，现金收费一般是从每个月 10 日开始的，托收账户的扣费也要在 11 日和 12 日这两天完成。我首先查看 v$session_wait 里的等待事件，老于直接启动了 topas，查看哪些进程在消耗 CPU 资源。在 v$session_wait 里，我看到了大量的 db file sequential read、db file scatter read 和 cache buffer chains 闩锁的等待。从这种情况来看，肯定是存在某个应用大量访问某些表，导致消耗了大量 CPU 资源。老于从 topas 上也看到了 10 多个消耗 CPU 资源十分严重的进程。老于立即通过 v$process 和 v$session 找到了这些进程正在执行的 SQL。我也通过 OEM 的 Top SQL 工具定位了这些 SQL，和老于找到的是一致的。情况很明确了，有一个 SQL 占用了过多的系统资源，

从而导致系统 CPU 资源被耗尽，导致 cache buffer chains 闩锁出现问题。这个 SQL 来源于 10 多个不同的会话，看样子是一个什么程序模块导致的，而不是由于某个人的个人操作，因此我们把 SQL 告诉小孙，让他联系开发人员，确认这个 SQL 究竟属于哪个应用模块。小孙通过电话询问了开发人员，很快就找到了原因，这个 SQL 属于一个批量打印发票的程序。我们几个把这个 SQL 分析了一下，它开销最大的部分是对某张几千万数据的大表通过索引扫描查找数据，每次会查出几万条数据，然后通过 Hash 连接的方式和其他表连接。从执行计划上来看，也没有明显不合理的地方，感觉短时间内还没有办法解决这个问题。这个应用不是简单地加个索引就能够优化的，里面包含的几个 SQL 都是五六张表的连接，想优化确实还不容易。

通过和营业厅沟通，了解到批量打发票一般是在每个月 10 日以后开始做，由于比较耗资源，所以这个应用一般都是晚上跑，白天是不允许做的。不过由于最近系统性能存在问题，这个月的出账晚了一天，所以这个程序 10 日没有跑，11 日晚上才开放了这个功能。为了赶时间，个别营业厅把这个业务放到白天来做了，这才导致了今天下午出现了系统性能问题。老于马上通知了张工，希望张工协调一下这些营业厅，让他们暂停这个操作。通过张工和这几个营业厅协商，这个应用暂停，压到 18：00 以后继续运行。得到张工确认后，我们将还在跑的 10 多个会话杀了，系统负载一下子降了下来，从 vmstat 上看，运行队列一下子降低到 20～40，看样子问题解决了。

问题解决后，张工打了个电话过来，老于正在洗手间里，所以我替他接了这个电话。在电话里，张工针对刚才这个问题感慨了半天，我也一直感觉张工绕来绕去的，肯定是有什么话想说又不好意思说。我就问他是不是有什么话不太方便说，他于是问我能不能提前做点什么，把目前我们已经分析清楚可以做的优化操作提前做一下，缓解现在的紧张局面。我说目前我们的主要任务还是分析问题，按照项目计划，我们的第一次实施时间是 6 月 15 日至 6 月 25 日，现在做点什么可能会起到一定的作用，但是对于整个优化项目来说，会把先前的计划完全打乱，甚至最终可能会影响整个项目，反而得不偿失。张工略感失望，但是也很无奈，毕竟一年都挺过来了，也就不差这十天半个月了。

本来下午想和开发厂商开个碰头会，由于刚才的突发事件，一直拖到快下班时才腾出时间来。不过见面的结果很令人失望，虽然应用厂商在这边留有一支近 10 个人的开发队伍，并且配备了一名项目经理，但是基本上帮不了什么忙。项目经理小王是个不到 30 岁的年轻人，挺不错的东北小伙子，不过他的职权有限，一切行动都必须听范总的。而且这个开发团队的主要工作是解决小版本升级的问题，最近手头的工作特别忙，主要精力都在应付需求变更，根本没有时间来配合我们做一些应用的分析与调整。为了这个项目，开发商安排了一个配合我们做优化的工程师，这个工程师就是平时和我们在一起工作的小孙，他目前的主要任务是负责应用运维，实际上充当的是一个服务台接线员的角色，负责接客服热线，然后把问题转发给现场的开发团队。作为刚刚毕业不到一年的新手，小孙基本上只能充当一个现场协调的角色，甚至连协调这种工作也不一定能够很好地完成。小孙和我所希望的能够配合我们进行 SQL 优化的人员相差甚远，不知道是开发商故意这么安排呢，还是他们没有理解我们的实际要求。这些问题看样子必须在明天的四方协调工作会议上提出，因为从初步的分析来看，SQL 调优将是项目成败的关键。本来我想了解一下应

用系统的架构以及他们对系统瓶颈的认识。不过开发商那边以最近任务比较紧并且现场工程师对这些情况不太了解为由拒绝了，看样子应用软件方面的问题，今后依赖开发商来提供信息不大现实，一切都要靠自己了。

今日点评

今天刚刚进场，所以老白主要是了解系统的状态，这需要从服务器、操作系统、存储、数据库等多个层面去了解。为此老白也准备了一个检查表，内容如下。

	检查项目	相关文件或结果	状　态	备　注
日志及文件	Oracle Alert日志			
	bdump/udump下是否存在明显的报警			
	listener相关日志			
	SQL*Net日志			
	参数/参数文件			
	listener.ora/tnsnames.ora			
操作系统	操作系统版本			
	检查操作系统补丁			
	节点名			
	操作系统vmstat状态			
	操作系统I/O状态			
	操作系统进程情况			
	操作系统异步I/O情况			
	系统交换区情况			
	主机物理内存情况			
	内存使用情况			
	CPU个数			
	CPU使用情况			
	网络情况： ❑ 网卡情况 ❑ 路由情况 ❑ hosts文件 ❑ netstat信息			
	文件系统情况，重点检查ORACLE_HOME、归档、temp等相关目录			
	内核参数和ulimit			
数据库配置	数据库版本，SQL*Net版本			
	数据库大小（所有数据文件和undo表空间文件的大小）			
	表空间的数量			

（续）

	检查项目	相关文件或结果	状　态	备　注
数据库配置	数据文件的数量			
	数据文件的属性			
	数据文件响应性能			
	表空间容量			
	重做日志情况 重做日志组数 重做日志文件大小 日志切换情况 归档情况			
	控制文件情况			
	回滚段配置			
数据库性能	数据库负载与主要命中率指标			
	DB Cache分析			
	共享池分析			
	PGA分析、排序情况分析			
	主要等待事件			
	闩锁分析			
	锁性能分析			
	undo性能分析			
	RAC性能分析			
应用性能	Top SQL			
	Top会话			
	应用的节奏			
	业务并发量			
其他采集项	备份策略			
	定期Job情况			
	停机窗口情况			
	应用体系架构			
中间件	中间件服务器情况 （配置、容量、架构等）			
	数据库连接池			

实际上，对于不同的系统，这个检查表是不同的，其目的就是对系统进行一次全面的体检，以便了解系统以及存在的问题。

今天除了了解系统情况外，还发生了一次小事故。由于某个业务在白天运行，导致 CPU 遇

到了瓶颈，最后决定停掉这部分应用，等晚上再跑，临时解决了问题。出现系统性能问题时，可以通过 Top 工具找到 CPU 消耗比较大的进程，然后通过 OEM 或者其他第三方工具对这些进程进行跟踪，就能够找到这些进程在做什么，从而找出故障原因。当然使用脚本 `v$process`、`v$session` 和 `v$sqlarea` 进行分析也能达到同样的目的。不过如果能用工具，为什么不用呢？老白是十分喜欢使用工具的。

今天碰到的性能问题，老白他们暂时无法解决，最后是通过和营业厅协商，让他们暂停了某个业务模块才解决的。性能优化过程中碰到的问题并不一定必须要通过技术手段来解决，实际上很多情况下可以通过和客户协商来解决问题。在系统出现了严重问题而暂时又没有很好的办法来解决时，暂停一些可以暂停的业务也是一种十分常用的解决方法，起码来说，停掉几个可以晚上跑的模块总比系统整个不能用要好得多。这一点，客户是可以理解的。

另外，今天老白拒绝了张工要求提前采取优化手段的要求，这一点体现了老白的老到。这种大型优化项目不是一朝一夕的事情，失败的案例也比比皆是。这种项目必须按部就班地进行，早期盲动是十分不明智的，必须有的放矢才能获得好的效果。老白他们刚刚进场，对系统根本不了解，这时候采取一些临时措施可能效果不大，甚至可能出现负面的效果，所以在这种情况下，老手都会尽可能避免过早动手。

优化小技巧　数据库优化的步骤

数据库优化是一个系统工程，需要有严密的组织和严格的步骤。这里介绍一种最常用的数据库优化步骤。一个完整的优化过程应该包括以下几点。

- 明确系统现状。
- 制订优化目标。任何优化都不可能是绝对彻底的优化，都只是针对目前系统状态的一种资源重组。优化的目标越高，优化的难度也越高，其成本也越大。因此制订合理的优化目标是十分重要的。
- 采集系统并分析数据。根据优化目标，对数据库进行数据采集，并对系统进行全面的分析，找到系统中的性能瓶颈。
- 制订优化计划。根据采集的数据和分析结果，制订优化方案和实施计划。
- 审核优化计划。和客户一起审核优化计划，在必要的情况下还需要和应用开发商一起进行优化计划的审核，确保优化计划是切实可行的。
- 实施优化计划。
- 监测优化系统。数据库优化是整体的优化，对于某些应用，优化后可能会有反效果。因此优化完成后，需要对所有的应用系统进行监测，如果有某些应用受到比较大的负面影响，应该尽快解决这些问题。
- 评估优化效果。优化完成后，对优化的效果进行一次评估。

这个优化过程不是一次完成的，如果评估优化效果时发现没有达到预期的目的，这个过程还需要从头再来一次，直到达到优化目标为止。

在进行数据库优化之前，首先要了解目前系统的运行状态和目前系统资源的使用情况，了解什么是目前系统中最大的问题，比如：

- 系统服务器的型号；
- CPU 的速度及数量；
- 内存大小；
- I/O 设备情况；
- 网络情况；
- 系统用户数量及并发用户数量；
- 目前系统的 CPU 使用率、内存使用率；
- 目前 I/O 系统状态（平均读写响应时间、I/O 等待时间等）。

然后要根据系统的现状，对于优化的目标进行定位。比如，优化（升级）后要达到：

- 平均系统响应时间在 X 秒以内，数据库响应时间在 X 秒以内；
- CPU 使用率保持在 80%以下；
- 系统内存使用率保持在 90%以下，不发生换页操作；
- XX 统计可以在 XX 分钟内完成；
- XX 操作，操作员的等待时间不超过 X 秒钟；

……

只有清晰了解了系统目前的现状，才有可能形成合理的优化目标，为整个优化工作打下一个坚实的基础。优化目标的设定对优化的成败至关重要，目标不是定得越高越好，而是要切合实际。数据库优化是一种高成本的工作，优化的目标越高，需要投入的成本也越高，量入为出，才能制订出合理的优化目标，从而指导整个优化工作顺利进行。

制订优化目标时，除了考虑成本因素外，还要考虑可以用于实施优化工作的各种资源以及它们如何配合。比如要做 SQL 优化，就需要有 SQL 优化的专业技术人员参与，更多的时候也要有应用开发厂商的配合。如果这些资源不能够协调好，那么 SQL 优化工作的成效就要大打折扣了。在考虑各种资源的时候，大家往往会忘记一个资源，那就是时间。任何一个优化工作都是有时间要求的，如何在规定的时间范围内达到优化的目的，需要制订优化目标的人充分考虑。

在优化工作中，存在各种各样的风险，会影响优化目标的实现。在制订优化目标时，把各种风险因素考虑进去是十分关键的。除了在制订优化目标时要考虑风险因素外，尽早地让其他相关部门了解风险的存在，提高其风险意识和抵御风险的能力，也是优化成败的关键。

优化小技巧　按部就班是成功的关键

今天张工提出了，我们能否提前做一些优化工作，为什么我会拒绝这个请求呢？系统优化是

一个周期较长的项目，而需要进行全面优化的系统一般来说都存在很严重的问题，客户希望你一进现场马上就给系统带来很大的改善，因此提出提前进行优化的要求也是很正常的。

不过由于对系统的了解是逐步深化的，在优化项目初期，我们对系统的认识比较片面，这个时候采取任何优化动作都存在较大风险。在这个阶段，不应该打乱原本的计划，提前采取任何优化措施。优化项目不仅仅是技术问题，它的涉及面十分广泛。一般来说，优化项目的发起者是IT部门，而对于绝大多数客户来说，业务部门才是考核的主体，他们的感受反馈到领导那里，分量会比IT部门要大得多。从业务部门来说，系统扩容的风险远比优化项目小，因此他们和IT部门往往是有矛盾的。优化过程中出现了某些变数，从而导致优化项目夭折的案例比比皆是。

因此作为优化实施团队，不盲目动作是十分关键的。我曾经参加过一个项目，优化工作开始后一个多星期，突然他们的运维部主任把我找了过去，很严厉地说，为什么我们做了优化，系统反而更差了。我告诉他说，我们目前主要还是处于采集数据的阶段，不会对系统做任何调整，所以越优化系统越差的说法不成立，这几天系统性能不好主要是由于业务部门在开展促销活动，所以业务量比平时高了不少。主任如释重负，说："只要不是你们搞的就好。可是，这几天业务量比平时高你怎么不通知我？害得我在局务会上又被大家攻击了。"

大多数有经验的DBA能够很好地控制优化项目的进程，完全按照自己的思路往前走，这样的项目控制能力对于项目的成功是十分关键的。而很多初次参与优化项目的DBA往往过于急躁，总想一下子就解决问题，或者尽快露一小手给客户看看。如果这样发展下去，项目中出现问题的机会就加大了。所以说，在一个优化项目中，成功地控制项目的进度，使之在自己的掌控下前进，对于项目成功是十分重要的。

优化小技巧　优化常用知识2：常用诊断事件清单

Oracle的内部诊断事件相当多，在一些复杂的诊断与优化工作中，DBA也经常会通过诊断事件去分析问题，或者通过设置诊断事件去解决某些bug。对于一个资深的DBA来说，需要在日常工作中不断地积累这方面的知识。下面这张表是老白收集的主要事件清单，不过有很多事件老白也没有用过，大家在使用时要十分慎重。

事　件	说　明	例　子
Event 10013 - Monitor Transaction Recovery	在Startup时跟踪事务恢复	ALTER SESSION SET EVENTS '10013 trace name context forever, level 1';
Event 10015 - Dump Undo Segment Headers-	在事务恢复后，做转储回退段头信息	ALTER SESSION SET EVENTS '10015 trace name context forever, level 1';
Event 10032 - Dump Sort Statistics	转储排序的统计信息	ALTER SESSION SET EVENTS '10032 trace name context forever, level 10';
Event 10033 - Dump Sort Intermediate Run Statistics	排序过程中，内存排序区和临时表空间的交互情况	ALTER SESSION SET EVENTS '10033 trace name context forever, level 10';
Event 10045 - Trace Free List Management Operations	freelists的管理操作	ALTER SESSION SET EVENTS '10045 trace name context forever, level 1';

（续）

事　件	说　明	例　子
Event 10046 - Enable SQL Statement Trace	跟踪SQL，有执行计划,绑定变量和等待的统计信息，Level 12最详细。 Level的定义如下。 Level 1：SQL语句，执行计划和执行状态。 Level 4：Level 1的信息加上绑定变量信息。 Level 8：Level 1的信息加上等待事件信息。 Level 12：Level 1＋Level 4＋Level 8	`ALTER SESSION SET EVENTS '10046 trace name context forever, level 12';`
Event 10053 - Dump Optimizer Decisions	在分析SQL语句时，转储出优化器所做的选择，Level 1最详细。 Level的定义如下。 Level 1：状态和估算信息。 Level 2：只显示估算信息	`ALTER SESSION SET EVENTS '10053 trace name context forever, level 1';`

Event 10060 - Dump Predicates	转储SQL语句中的断语信息。要在需要转储的用户下创建下表：
	`CREATE TABLE kkoipt_table(c1 INTEGER,`
	`c2 VARCHAR2(80)`
	`);`
	断语信息会写入该表
		`ALTER SESSION SET EVENTS '10060 trace name context forever, level 1';`

Event 10065 - Restrict Library Cache Dump Output for State Object Dumps	限制对象状态转储的时候，库缓存信息的详细程度。 Level的定义如下。 Level 1：只包含library对象的地址信息。 Level 2：Level 1的信息加上library对象锁的信息。 Level 3：Level 2的信息加上Library对象及对象句柄。 默认是Level 3	`ALTER SESSION SET EVENTS '10065 trace name context forever, level level';`
Event 10079 - Dump SQL*Net Statistics-	转储SQL*Net的统计信息	`ALTER SESSION SET EVENTS '10079 trace name context forever, level 2';`
Event 10081 - Trace High Water Mark Changes	跟踪高级水印（HWM）的改变	`ALTER SESSION SET EVENTS '10081 trace name context forever, level 1';`
Event 10104 - Dump Hash Join Statistics	Hash连接的统计信息	`ALTER SESSION SET EVENTS '10104 trace name context forever, level 10';`

（续）

事　件	说　明	例　子
Event 10128 - Dump Partition Pruning Information	分区表调整信息。在9.0.1或者后面的版本，在Level 2后还需要建立如下的表： CREATE TABLE kkpap_pruning (partition_count NUMBER, iterator VARCHAR2(32), partition_level VARCHAR2(32), order_pt VARCHAR2(12), call_time VARCHAR2(12), part# NUMBER, subp# NUMBER, abs# NUMBER); Level取值： 1　Dump pruning descriptor for each partitioned object 0x0002　Dump partition iterators 0x0004　Dump optimizer decisions about partition-wise joins 0x0008　Dump ROWID range scan pruning information	ALTER SESSION SET EVENTS '10128 trace name context forever, level level';
Event 10200 - Dump Consistent Reads	转储一致读的信息	ALTER SESSION SET EVENTS '10200 trace name context forever, level 1';
Event 10201 - Dump Consistent Read Undo Application	转储一致读涉及Undo信息的内容	ALTER SESSION SET EVENTS '10201 trace name context forever, level 1';
Event 10220 - Dump Changes to Undo Header	转储出Undo头信息的改变	ALTER SESSION SET EVENTS '10220 trace name context forever, level 1';
Event 10221 - Dump Undo Changes	转储Undo的改变	ALTER SESSION SET EVENTS '10221 trace name context forever, level 7';
Event 10224 - Dump Index Block Splits / Deletes	转储索引块的分裂和删除信息	ALTER SESSION SET EVENTS '10224 trace name context forever, level 1';
Event 10225 - Dump Changes to Dictionary Managed Extents	转储字典管理的扩展变化	ALTER SESSION SET EVENTS '10225 trace name context forever, level 1';
Event 10231-Ship Corrupted Blocks on Table Scans	全表扫描时跳过坏块，在有坏块的情况下做数据拯救时很有用	ALTER SYSTEM SET EVENTS '10231 trace name context forever,level 10';
Event 10241 - Dump Remote SQL Execution	远程SQL语句的执行信息	ALTER SESSION SET EVENTS '10241 trace name context forever, level 1';

（续）

事　件	说　明	例　子
Event 10246 - Trace PMON Process	跟踪PMON进程。只能修改参数，不能用ALTER SYSTEM	event = "10246 trace name context forever, level 1"
Event 10248 - Trace Dispatcher Processes	跟踪Dispatcher进程的工作情况	event = "10248 trace name context forever, level 10"
Event 10249 - Trace Shared Server (MTS) Processes-	跟踪共享服务器进程的工作情况	event = "10249 trace name context forever, level 10"
Event 10270 - Debug Shared Cursors	调试共享游标	event = "10270 trace name context forever, level 10"
Event 10299 - Debug Prefetching	跟踪表数据块和索引数据块的Pre-fetching	event = "10299 trace name context forever, level 1"
Event 10357 - Debug Direct Path	调试直接路径	ALTER SESSION SET EVENTS '10357 trace name context forever, level 1';
Event 10390 - Dump Parallel Execution Slave Statistics	转储并行操作中从属进程（slave）的统计信息	ALTER SESSION SET EVENTS '10390 trace name context forever, level 1;
Event 10391-Dump Parallel Execution Granule Allocation	转储并行操作的粒度	ALTER SESSION SET EVENTS '10391 trace name context forever, level 2';
Event 10393 - Dump Parallel Execution Statistics	转储并行操作的统计信息(每个从属进程单独列出）	ALTER SESSION SET EVENTS '10393 trace name context forever, level 1';
Event 10500 - Trace SMON Process	跟踪SMON进程	event = "10500 trace name context forever, level 1"
Event 10511 - turn off SMON check to cleanup undo dictionary	关闭SMON检查并清理undo数据字典	event="10511 trace name context forever, level 1"
Event 10513 - disable transaction recovery	关闭延迟的事务恢复	event="10513 trace name context forever, level 2"
Event 10608 - Trace Bitmap Index Creation	跟踪位图索引创建的详细过程	ALTER SESSION SET EVENTS '10608 trace name context forever, level 10';
Event 10704 - Trace Enqueues	跟踪锁的使用情况	ALTER SESSION SET EVENTS '10704 trace name context forever, level 1';
Event 10706 - Trace Global Enqueue Manipulation	跟踪全局锁的使用情况	ALTER SESSION SET EVENTS '10706 trace name context forever, level 1';
Event 10708 - Trace RAC Buffer Cache	跟踪RAC环境下的Buffer Cache	ALTER SESSION SET EVENTS '10708 trace name context forever, level 10';
Event 10710 - Trace Bitmap Index Access	跟踪位图索引的访问情况	ALTER SESSION SET EVENTS '10710 trace name context forever, level 1';
Event 10711 - Trace Bitmap Index Merge Operation	跟踪位图索引合并操作	ALTER SESSION SET EVENTS '10711 trace name context forever, level 1';
Event 10712 - Trace Bitmap Index OR Operation	跟踪位图索引或操作情况	ALTER SESSION SET EVENTS '10712 trace name context forever, level 1';
Event 10713 - Trace Bitmap Index AND Operation	跟踪位图索引与操作	ALTER SESSION SET EVENTS '10713 trace name context forever, level 1';
Event 10714 - Trace Bitmap Index MINUS Operation	跟踪位图索引MINUS操作的情况	ALTER SESSION SET EVENTS '10714 trace name context forever, level 1';

（续）

事　件	说　明	例　子
Event 10715 - Trace Bitmap Index Conversion to ROWIDs Operation	跟踪位图索引转换为ROWID操作的情况	ALTER SESSION SET EVENTS '10715 trace name context forever, level 1';
Event 10716 - Trace Bitmap Index Compress/Decompress	跟踪位图索引压缩和解压缩情况	ALTER SESSION SET EVENTS '10716 trace name context forever, level 1';
Event 10717 - Trace Bitmap Index Compaction	诊断位图索引	ALTER SESSION SET EVENTS '10717 trace name context forever, level 1';
Event 10719 - Trace Bitmap Index DML	跟踪位图索引列的DML操作（引起位图索引改变的DML操作）	ALTER SESSION SET EVENTS '10719 trace name context forever, level 1';
Event 10730 - Trace Fine Grained Access Predicates	跟踪细粒度访问的断语	ALTER SESSION SET EVENTS '10730 trace name context forever, level 1';
Event 10731 - Trace CURSOR Statements	跟踪游标语句。Level的定义如下： 1——显示父查询和子查询。 2——只显示子查询	ALTER SESSION SET EVENTS '10731 trace name context forever, level level';
Event 10928 - Trace PL/SQL Execution	跟踪PL/SQL执行情况	ALTER SESSION SET EVENTS '10928 trace name context forever, level 1';
Event 10938 - Dump PL/SQL Execution Statistics	转储PL/SQL执行状态。使用前需要执行rdbms/admin下的tracetab.sql	ALTER SESSION SET EVENTS '10938 trace name context forever, level 1';
flush_cache	刷新Buffer Cache	ALTER SESSION SET EVENTS 'immediate trace name flush_cache';
DROP_SEGMENTS	手工删除临时段。当这些临时段无法自动清除时，可以手工清除。ts#是指要删除临时段的表空间的ts#	ALTER SESSION SET EVENTS 'immediate trace name DROP_SEGMENTS level ts#+1';

5月13日 联席会议

今天是星期六，但是为了赶项目进度，我们三方也没有休息就召开了三方联席工作会议。说是三方，实际上是四方，除了甲方、应用开发商和我们外，北方公司 IT 部的领导也通过电话会议的方式参与了部分讨论。

老方和小齐是昨晚坐火车从北京赶过来的，早上一下火车就立马赶到位于浑南的总部。虽然之前已经在好几个项目上和小齐、老方合作过，但是我们都是通过电话和电子邮件沟通。正式面对面和他们打交道还是第一次。之前和老方通过几次电话，不过我两年前就听说过他这个人了。那时我受邀参加了海尔售后服务系统（HCSP）优化的项目，那是一个由 HP 总集成的项目。HCSP 系统自从上线后一直存在性能问题，到了 2004 年 7 月，性能问题已经导致了系统无法继续上线新的网点，因此必须尽快解决性能问题，才能确保系统按时上线。我入场后看到的第一个文档就是老方写的关于 HCSP 系统的优化建议。文档写得比较简略，大概不到 20 页，不过文档中提到的主要问题以及解决问题的方法，基本上和我经过几天分析后的想法一致。我当时就问客户，看这个文档，我感觉那个人基本上抓住了问题的要点，如果这么实施下去，应该可以解决问题，为什么没有让他继续完成这个项目呢，因为我来做这个项目，方向上不会有太大的变化。客户说他也不太清楚，那个人来了一次，就不愿意再来了。这让我感到十分诧异，有钱都不愿意赚的人还是比较少的。后来过了不多久，就听说老方加盟了 HP 公司，在 HP 公司的高级服务团队里做了负责 Oracle 优化服务的售前。

开发厂商来了一个重量级的人物——范总，他以前是这个项目的项目经理，现在已经是这个产品的产品线经理了。由于这个项目是范总亲自带队完成的，因此他对目前客户系统的情况十分清楚，孙主任也一再要求他亲自担任优化项目开发方的负责人。这种联席会开得不好就会成为务虚会，这也是我最担心的。从人员上，开发商参与会议的包括范总和我们认识的小孙，我们这边人到得很齐，除了三个现场项目实施的，项目经理小齐和总协调老方也都赶到现场，客户这边负责 IT 维护的孙主任和张工他们悉数到场。

会议从一开始就开得很热烈，气氛也很好，大家都是豪言壮语，谈笑风生。由于这种会议没有什么压力，所以大家都表现出幽默和机智。不过我自己觉得，在会议过程中收获的东西太少，至少从应用开发厂商与会代表的慷慨陈词中，我感受到了很大的压力。因为以往经验告诉我，在这个项目里应用开发厂商的配合肯定不会很好。这次会议把客户、开发商、优化厂商之间的职责、接口、升级流程都确定了下来，也确定了每个流程的里程碑以及责任人。我在这个项目中总体负责应用优

化，所以在讨论前面一些问题时，我一直没有开口。而谈到接口时，我一再强调我们不直接和开发商接口，我们的优化建议必须通过客户确认，然后由客户直接下达给开发商。这个要求一提出，客户和开发商都提出了反对意见，觉得这是多此一举。刚开始讨论这个问题时，开发商认为这个接口应该是我们和开发商之间直接的接口，不需要经过甲方。甲方的意思也是这个项目我们和集成商都是参与方，客户对双方都是支付了费用的，所以客户要的就是最后的优化效果，至于中间我们和集成商如何协商，那就是我们自己的事情了，如果还要通过他们转一下，必定会影响效率。

我坚决反对这种接口模式。以往的经验告诉我，集成商可以买客户的账，但是绝对不会买我们的账。在优化项目中，我们的工作成果越大，开发商的工作量也越大，而开发商是不会对优化项目的成败负责的，因此他们在这个项目中做的事情越少越划算。最早理解我的想法的是孙主任，他很快就明白我为什么坚持这种接口模式，所以孙主任很快就改变了开始的态度，转而支持我了。当我和孙主任反复坚持这种接口模式时，小齐和老于还感到有点惊讶，觉得在这方面我似乎大惊小怪了。很快，老方也明白了我的意思，坚决站在我这边了。最后在我们三个的一再坚持下，应用开发商同意了这种接口模式。而老方更绝，当客户和开发商同意后，老方一再强调今天的会议纪要应在会后马上打印出来，几方一定要在会议纪要上签字。

由于折腾着打印签字，本来 12：00 准时结束的会议被拖到了 12：30。中午是小齐请客，他们吃完饭还要赶回北京向北方公司领导汇报，所以吃饭就安排在北站的“新洪记”。由于人太多，分开坐了两辆出租车，老于他们三个人先走了一步，我和小齐、老方坐了一辆出租车。在车上，我把自己的担忧告诉了小齐：“我觉得开发商不会十分配合这个项目，我们可能要面临孤军奋战的不利局面。云南联通那种情况在沈阳不会重演。”云南联通是小齐做的第一个优化项目，实施是我做的，当时由于应用软件还没有验收，开发团队还在客户现场做验收阶段的工作，因此和我们配合得十分好。我们指出存在问题的 SQL，他们马上安排开发人员和我们一起分析和解决问题。看样子小齐对目前这种情况并无思想准备，所以听到我这么说之后就有点着急，掏出电话就想直接找北方公司领导协调。老方制止了小齐，并对我说：“老白，你性子太急，这话我准备在火车上和小齐说的。这个问题我也看出来了，而且十分严重，但是目前找领导还不是时候。因为这个问题目前还只是我们的猜测，虽然这个猜测成立是 100%的，我们甚至要做好项目因此被迫延迟的思想准备。”小齐一听老方说项目会延期，当时就急了。由于这个项目是样板项目，关系到后续其他几省合同的问题，因此不能延后，否则老板那边交代不过去。我只好安慰小齐说：“没关系，再困难的项目我们也见过，这里再困难也不会有海尔困难吧，那个项目我们不也拿下来！”

饭桌上没有再谈这个话题，这是我建议的，老于的压力已经够大了，不能再给他施加压力了。在饭桌上我问了老方，为什么海尔的项目他已经做到那种地步，反而退出了。老方告诉我，当时是有个高人建议他退出的，因为客户、原厂和集成商三家打得不可开交，集成商（也就是软件开发商）基本上采取对抗的态度，所以他觉得这个项目很难推进，弄不好把一世英名都搭进去了，不划算。想想我当年如履薄冰的样子，确实也像老方所说的情况。做 DBA 的，想干成功一百个项目容易，想不干砸一个项目却很难，大家可能不会长久记得你的成功，而你的每一次失败都会被永久地流传在这个行业里，确实需要像鸟儿爱惜羽毛一样爱惜自己的声誉，因为这关系到你的

饭碗。老方也很佩服我当时敢把那个地雷接到手里，这个不可能完成的任务在我之前还有其他好几个 DBA 曾经去过，都是去了一次就决定退出了。

海尔项目的成功有很多因素。当年做海尔项目时，我在优化方面的功底并不深厚，只是做过一些小型的优化项目。敢于第一次出手就拿这样没谱的项目试刀，我的那种车到山前必有路的信念是支撑我完成这个项目的关键。很多次山穷水尽的时候我都坚持下来了，有些手段现在想来也算不上正道，而且成功也存在很多偶然因素，不过我确实取得了令人不可思议的成功。

由于是中午，不能喝酒，所以大家要了一大瓶可乐，点了几个东北菜，当然少不了“新洪记”最有特色的水饺。大家基本上没怎么谈项目的事情，由于没有喝酒，老肖的压力比较小，所以话也较多。这些人里，除了小齐比较年轻，其他人都在 IT 圈里混了 10 多年了，所以大家聊的都是 Oracle 圈子里的一些往事，甚至聊到了当年的北京巨龙，那批中国搞 Oracle 的先驱，其中大多数人都已经功成身退，甚至很多人已经移民海外了。老肖当年是甲方，是巨龙最大的客户，那场轰轰烈烈的 Oracle 状告巨龙的官司是由老肖他们单位引起，最终也是由老肖他们单位出面摆平的。说起这段往事，让人确实觉得圈子太小，通过 6 个人可以在地球上任何两个人之间建立联系，“六度空间”之说确实不虚。谈到了当年巨龙的成功与失败，大家都唏嘘不已。20 世纪 90 年代中期崛起的这个中国最早的 Oracle 第三方服务厂商，其中绝大多数人员都是从 Oracle 出来的，销售能力、技术力量都不逊色于当年的 Oracle 中国区。在那个版权意识尚处于萌芽状态的时期，巨龙的盗版软件的概念也吸引了大批既需要上大型数据库系统，又不希望出太多银子的客户。他们是有机会成为中国的 EDS 的，不过最终他们并没有成功，主要原因还是因为分裂，最终巨龙分裂成几家小公司，这些小公司再也没有像巨龙一样辉煌过。一个团结的巨龙可以成功，而分裂只能带来一些平庸的小公司，它们在历史的长河里跃动了几下，就像泡沫一样消失在浪花里了。

午饭后，老方和小齐就动身回北京了。今天是周六，下午没什么事情了，我和老肖准备回去休息，老于觉得不放心，坚持要去机房看看系统。于是大家约好晚上一起喝点酒，我和老肖就回宾馆了。沈阳有太多的朋友，所以不敢轻易露头，一旦露了一小脸，那无穷无尽的酒会就接踵而至了。所以我只打电话通知了老张，老张是我从小一起长大的朋友，情同手足，他本来在外地做项目，正好今天晚上回来，于是约好周日晚上一起去喝酒。他又叫上他的几个朋友，他们都是我认识的。和他圈子的朋友喝酒相对还是比较安全的，不会牵出太多的社会关系，这样被放倒的次数也就大大减少了，这是我回沈阳的一贯做法。

今日点评

在优化项目中，如何和开发商或者集成商配合是个难题，因为优化不可避免地会涉及一些数据库以外的东西。比如，要调整底层存储的配置，需要修改一些存在问题的 SQL。而这部分调整的主动权不是掌握在我们手里，因此老白不希望直接和开发商接触，在这种情况下，我们一定和甲方站在一起去面对开发商，因为甲方手里往往都有制约开发商的手段，而我们没有。如果我们直接和开发商接触，开发商不想配合的话，是很难推进的。因此在优化项目中，如果发现开发商很难缠，那么就要尽量避免直接面对开发商，优化方案要通过开发商提交给甲方，并由甲方负责

监督优化进度。当然在优化过程中，和开发商直接沟通是有必要的。比如，开发商需要你向他们介绍你的方案，那么你就要尽可能地满足他们的要求。当然也可能在有些情况下，开发商的配合十分好，那么直接面对开发商，效率要高得多。

由于沈阳这个项目甲方是十分积极的，他们对项目成败的渴望并不亚于老白他们自身，所以甲方是值得信赖和依靠的。在这种情况下，老白选择了通过甲方将任务下达给开发商的方式，这个方式虽然看起来效率不高，不过却是此时唯一的选择。如果由老白他们直接和开发商沟通，那么会产生更多的摩擦，效率可能更为低下。

优化小技巧　优化流程

1. 数据采集

制订完优化目标后，需要进一步对系统中的数据进行采集和分析。数据采集包括操作系统数据和数据库的数据。对于不同的系统状态和不同的优化目标，采集的数据也有所不同。

对于操作系统数据的采集，可以使用 vmstat、top、sar、iostat、netstat 等工具进行。对于不同的操作系统，其命令可能略有不同。对于 HP-UX，glance 是不错的性能分析工具，在 AIX 上可以使用 nmon。Oracle 提供了一个很不错的 OS 监控工具 OSW（Metalink Nodes 301137.1），也可以使用 OSW 来对 OS 进行监控。

对于数据库数据的采集，可以通过多种方式进行。首先，进行一次 STATSPACK 分析（对于 Oracle 8 以前的版本，可以使用 bstat/estat 工具采集数据；对于 Oracle 10g 或者更新的版本，可以使用 AWR 报告、ADDM 报告和 ASH 报告）。根据 STATSPACK 报告，可以分析出很多数据库目前的状态信息。根据这些情况，再进行进一步的分析。如果你使用的是 Oracle 10g 或者更高的版本，还可以通过 ASH 来分析会话的情况。

Toad 也是一个数据库状态采集很有效的工具。通过 Toad 中的健康检查工具，可以找出目前系统中的不健康因素。通过 SGA 工具，可以定位开销比较大的 SQL 语句。

OEM 中的**诊断包**和**调优包**是采集数据库状态信息的很有效的工具。使用 OEM 的实时监控工具的挖掘功能，可以很方便地进行问题定位。如果是 Oracle 10g 的数据库，那么可以直接使用 ADDM 去分析数据库的性能。

很多 DBA 都有一些采集数据的独门秘笈，根据不同的数据库的情况，准备一些数据库性能状态采集的脚本是十分必要的。作为性能优化人员，应该准备以下方面的数据库脚本（部分脚本的功能可以由 OEM、Spot light 等工具实现）：

- 数据库主要表、索引、分区、表空间等情况的信息采集脚本；
- 数据库表的主要约束关系情况的采集脚本；
- 获取某个数据库对象的 DDL 语句的脚本；
- 数据库参数的采集脚本（包含隐含参数）；
- 主要缓冲区命中率情况的采集脚本；
- SQL 缓冲区分析脚本；

- 获取长时间运行的 SQL 脚本；
- 共享池分析的脚本；
- 系统等待事件分析的脚本；
- 闩锁竞争情况采集脚本；
- 主要文件 I/O 性能分析的脚本；
- Top 会话分析的脚本；
- 会话跟踪的脚本；
- OPS/RAC 性能分析的脚本（可以使用 Oracle 提供的分析脚本，在?/rdbms 目录下）；
- 表空间碎片分析的脚本。

部分脚本可以在 rdbms 目录下找到。另外，Metalink 上也有大量脚本可以参考。平时积累一些有用的数据库分析和数据采集脚本可以大大提高数据采集的效率。每个 DBA 都应该根据自己维护的数据库的特点，积累自己的“独门秘笈”。

采集数据是一个十分严密的过程。如果这个阶段采集的数据不准确，那么会导致优化计划的针对性不强，从而导致优化失败。因此，应该在数据库的不同状态点（或者时间点）进行多次数据采集，而且相同的状态点（或者时间点）最好进行多次采集，并进行对比，找出共同点。如果发现某个状态点两次采集的数据差别比较大，那么就需要继续采集数据，直到所有的数据库状态都可以被清晰地分析出来为止。

另外，在采集数据的时候，要注意像交响乐一样，每个系统都是有自己独特的“节奏”的，这里所说的“节奏”其实就是每个系统独有的运行特征。每个系统的闲时、忙时、业务高峰都有一定的特征，在某个时间段执行的程序也有一定的规律。作为优化团队，在采集数据之前，就应该通过甲方或者集成商、应用开发商了解这个系统的“节奏”，以便于制定有针对性的数据采集计划。甚至更严格地说，系统的节奏会决定整个优化项目进场的时间。如果进场时间未能根据系统的节奏进行安排，我们甚至可能白白浪费时间。比如说一套电信的计费账务系统，这个系统的特点是平时这套系统的业务量比较平均，不会引起明显的性能问题，但是每个月的月底到月初这段时间，由于出账、集中销账等业务的存在，可能导致系统负载大幅度增长。因此，针对这种系统，我们的数据采集一定要在 25 日之前开始，持续到业务高峰结束的下个月 10 日左右。在安排这样的项目的时候，我们一定要在 20 日左右就能够进场，如果错过了 30 日到下个月 5 日这段业务最高峰的时间，可能就会浪费一个月的时间。

2. 优化方案设计

所有的数据库和系统状态都已经采集完毕，并确认数据准确以后，就需要对这些数据进行进一步的分析，通过分析制订出优化计划。这时，可能会发现需要采集更多数据才能够定位问题，因此就要进行数据补充，直到所有数据都能够分析清楚，并形成有针对性的优化方案为止。

制订优化计划一般来说无法由一个人完成，而要由一个专家团队来完成。某方面的专家可以提出自己对某个专题的计划，最后汇总为一个统一的计划。优化计划形成后，需要由一个专业的技术小组来审核该计划。专业小组应该包括下列人员：

- 应用开发方的技术代表；
- 小型机（服务器）技术专家；
- 网络管理员；
- Oracle 专家和 DBA；
- 系统架构师；
- 用户代表。

一份优化计划应该包括：

- 系统软硬件及网络调整和数据库调整方案；
- 优化方案涉及的所有脚本；
- 详尽的系统优化操作时间表，需要列出每个步骤的起始和终止时间；
- 优化完成后的检查方案（含脚本）；
- 优化过程出现问题的处理预案。

在制订优化实施计划时，一定要考虑保留足够的时间进行必要的备份和保留系统检查及回退的时间。对于有一定风险性的操作，一定要进行备份，或者保证有可用的备份集可以用于恢复，并且在近期有系统的全备份，这可以保证恢复的时间。一旦优化失败或者遇到一些问题，可能需要进行回退操作或者其他处理，如果没有预留这部分时间，在优化过程中发生问题时将没有足够的时间进行处理，从而影响系统的正常运行。

实施计划制订完成后，一定要经过专家小组的审核。参与的专家一定要对实施计划的每个细节进行审核，并且对实施计划中的时间表进行确认，确保时间表是可行的。

优化工作不可能 100%顺利完成，而 DBA 的责任是保障在任何情况下，系统都能够正常运行。因此，优化计划应该包含足够的应急预案，以应对在优化工作中碰到的各种异常情况。对于难度比较大的优化工作来说，应急预案是优化工作的有利保障。应急预案中应该考虑以下情况：

- 优化工作进度迟缓，无法按时完成；
- 发现优化工作无法达到预期目标，甚至可能引起新的性能问题；
- 数据库补丁无法正常完成；
- 数据库无法正常工作；
- 数据库崩溃；
- 系统硬件发生故障；

……

所有和本次优化相关的可能出现的问题，都应该根据出现的概率和危害程度制订相关的应急预案。对于危害性越大的情况，应急预案也应该越完备。千万不可有侥幸心理。我参加过很多优化项目，在做优化实施时，回退的比例起码占到 5%左右。有时候由于时间很紧急，回退还是不回退成为十分艰难的决策。我曾经看到过优化过程中出现了一些问题，在选择是否回退时，决策者选择了不回退，最后导致第二天上午系统出现了严重的性能问题。

3. 实施优化计划

优化计划的实施一般来说会安排在晚上或周末进行。有些优化不需要停机进行，而有些优化需

要停止或者重启数据库，有些优化需要消耗相当长的时间（比如表或索引存储结构重组）。

实施优化计划前，需要向项目组提出申请，并由项目组协助安排优化时间。在完成必要的备份等准备工作后再进行实施。实施过程中需要随时对进度进行审核，如果发现实际的进度要远远慢于预案的进度估算，甚至可能发生无法在规定停机时间内完成的情况，就需要及时调整方案甚至及时停止实施，比较严重的情况还需要根据应急预案立即恢复。

在实施优化计划时，最好整理一个实施检查表（checklist），在实施工作中，每完成一个步骤就在该检查表上进行记录。当要执行某个优化步骤时，先检查检查表上的其他相关步骤是否已经完成。当优化工作完成时，还要查看检查表是否所有的步骤都已完成。

4. 优化后评估

优化方案实施后，应该安排 DBA 对数据库进行 24 小时～72 小时的实时观察（根据系统规模不同，实时观察的时间长短也不同）。观察的内容包括系统级和应用级两个方面，系统级观察包括：

- 系统 CPU 使用情况；
- 系统 I/O 情况；
- 系统内存情况；
- 系统网络状态；
- 数据库总体负载情况；
- 数据库平均 SQL 响应时间；
- 关键业务 SQL 的执行计划；
- 数据库平均事务响应时间；
- 主要等待事件；
- 数据库 I/O 情况。

除了观察系统状况外，还需要对应用系统进行观察，查看相关应用系统总体的工作状态是否正常。在这里要强调的一点是，优化实际上是一种系统变更行为，可能存在一些不确定的因素。因此，在优化实施之前，应该将对系统影响较大的 SQL 以及关键业务的主要 SQL 的执行状态和执行计划都采集下来，优化完成后，对这些 SQL 的执行情况进行比对。如果发现优化后某些 SQL 的平均每次执行开销明显大于优化前，那么就需要比对这些 SQL 的执行计划，找出故障原因，否则这些问题可能导致第二天出现灾难性的后果。

完成优化工作后，需要对优化的系统进行评估以确定优化是否达到目标。通过和优化前采集的数据进行比较，可以得到系统总体性能提升的幅度。优化效果评估需要严格按照优化方案中的优化目标进行。如果某个项目没有达到预期目标，需要找到原因，并确定是否需要进行下一步优化。

下面是一个系统优化前后比较的例子。

指标名	旧系统情况	当前情况	变化对比	备　注
系统CPU	30%~60%	60%~90%	比以前加大30%	CPU使用率明显提高
系统内存	90%	90%	0%	变化不大
系统I/O等待	40%~50%	0~30%	大幅度减少	目前基本处于正常状态
响应时间	11.47s	3.18s	加快3.37倍	
事务数/秒	3.51	9.38	提高2倍多	

一般来说优化不是一次就能够完成的，上述过程需要进行多次循环，直到所有的优化目标都已经达到才停止。在这个过程中，也需要对优化目标进行一些修正，如果最初设定的目标实现起来存在风险或者根本无法达到，那么就需要尽快修正优化目标。

优化小技巧　转储文件内容 1：数据文件

转储数据文件、重做日志等文件的内容也是优化中经常用到的技术，因此 DBA 必须掌握。在优化项目中，有时候为了分析表和索引等的碎片、行迁移等情况，需要对数据文件进行转储。把数据库文件（数据文件、日志文件、控制文件等）的内容取出有助于对 Oracle 数据库内部结构展开分析。使用操作系统的转储工具可以导出数据库文件中的内容，不过这种方式相对比较麻烦。Oracle 提供的 ALTER SYSTEM DUMP 命令就可以把数据导出到 Trace 文件中，这一命令是以数据块为单位的。

对于普通的数据文件，通过下面的命令可以导出数据文件中的数据块：

```
ALTER SYSTEM DUMP DATAFILE [<file_id>|<file_name>] block <block_no>;
```

或者：

```
ALTER SYSTEM DUMP DATAFILE  [<file_id>|<file_name>] block min <block_no_min>  block
max <block_no_max>;
```

第一条命令导出一个数据块，第二条命令可以导出多个连续的数据块。以上命令不能转储 LMT 的临时表空间文件。如果用 `ALTER SYSTEM DUMP DATAFILE` 命令去导出一个 LMT 的临时表空间文件，会报 ORA-1205 错误，具体如下：

```
SQL> ALTER SYSTEM DUMP DATAFILE 'E:\ORACLE\ORADATA\ORC9\sk_TEMP.DBF' BLOCK 1;
     ALTER SYSTEM DUMP DATAFILE 'E:\ORACLE\ORADATA\ORC9\sk_TEMP.DBF' BLOCK 1
     *
ERROR at line 1:
ORA-01205: not a data file - type number in header is 6
```

查看 ORA-1205 的错误信息：

```
Text:   not a datafile - type number in header is <num >
------------------------------------------------------------------------------
Cause:  The file type in the header is not correct for a datafile.
        This is probably a redo log file or control file.
Action: Restore a copy of the correct datafile and perform recovery as needed.
```

要转储 LMT 的临时文件，需要使用 `DUMP TEMPFILE` 命令：

```
SQL> ALTER SYSTEM DUMP TEMPFILE 'E:\ORACLE\ORADATA\ORC9\sk_TEMP.DBF' BLOCK 1;
     System altered.
```



聚　　会

今天是周日，没有安排工作。到沈阳几天了，一直都很忙碌，今天难得清闲，所以我一个人去故宫转了转。沈阳故宫是沈阳旅游业的王牌，它是满清入关前最后的皇宫，也是目前保留较为完整的、仅次于北京故宫的宫殿群。在我的印象里，故宫是一个值得一玩的地方，不过这一次我看到的却是满目的破败，完全没有了我小时候游故宫的样子。记得当年还写过一篇游故宫的作文，里面的沈阳故宫是金碧辉煌的，和眼前的千疮百孔完全对不上号。故宫中有三分之一的建筑没有开放，而开放的建筑也大多数破败不堪，甚至十王亭有几个亭子里居然堆放着水泥。睹物思情，想到昔日辉煌的皇家宫殿落魄到如此田地，不免有些伤感。

晚上约了 7 个哥们儿，都是老张的同学，我和老张同校但不同班，他们和我比较合得来，每次回沈阳，都会一起喝酒。照例是在“新洪记”啤酒馆，今天孙大夫、周秘书和大哥都来了，所以气氛也比较热烈。

孙大夫是外科医生，这要在深圳属于高收入人群了，但是由于他不愿意同流合污，因此日子过得比较清贫，但是他很满足。今天一来就大骂他女儿的老师，居然公开向他要红包，被他严词拒绝了。大家都说他太迂，劝他不要太认真了，为了这种事情和老师较劲，最后吃亏的是他女儿。大家说了一大通，还是没能说服他。按照孙大夫的意思，如果大家都去同流合污，那么社会风气永远也好不了，他哪怕做出点牺牲，也要给大家开个头。对于孙大夫的勇气，大家向来是比较钦佩的，于是大家嘻嘻哈哈的，又猛灌了孙大夫几杯。

周秘书是市领导的秘书，平时也比较忙，不过我和老张都回沈阳了，再忙也要出来舍命陪君子。长期跟领导出去，为领导遮风挡雨，练就了一身好酒量。而且周秘书知道的掌故颇多，每次聚会都会摆龙门阵，因此有周秘书在场，大家起码也能多喝两瓶。

今晚大家都没什么事，所以喝得比较放松，一大桌子菜没怎么动，大家都忙着频频举杯。沈阳人喝酒很豪迈，在广东一个人能喝一两瓶啤酒就算可以了，而在沈阳喝酒都是按箱算的。有一次过年，我刚买好了一箱啤酒，准备过年喝。来了一个同学，中午两个人就着我妈做的凉菜喝酒，我那个哥们把我这一箱啤酒全部干掉，还没喝过瘾。这次这顿酒从晚上 6 点多开始，一直喝到快 12：00 了，大家才算酒足饭饱。一共喝了 6 箱啤酒，平时不太喝酒的我今天也喝了四五瓶。

买完单出来，周秘书看样子还没尽兴，问还要搞什么活动不，大哥说算了，明天还要上班，大家就各回各家吧。我和周秘书、孙大夫同路，就一起打了个车。大家喝的都有点多了，上车后，周秘书习惯性地坐在前排，我和孙大夫坐在后排，我们两正在聊女儿上学的事情，突然周秘书把

手机递给我，说："老白，有熟人找你说话。"我一接电话，老洪那独有的声音就传了过来："老白，你也太不够意思了，到了沈阳都不给老哥我打个招呼，麻溜的，到我家边上的北京羊蝎子，我先过去等你们，周秘书知道那地方。"

老洪是我的学长，大学毕业后早就放弃了技术老本行，进入仕途，而且走得一直很顺，三十出头就高居处座，目前在沈阳做正处级办公室主任。老洪为人比较豪爽，给人大大咧咧的感觉，而实际上做事滴水不漏，是个当官的好料子。我们赶到北京羊蝎子时，老洪已经叫了一个羊蝎子火锅，几样小菜，自己坐在那里喝呢。这里只要要一个羊蝎子火锅，啤酒一律免费，能喝多少就送多少。

老洪见到我们，没再提我到沈阳没打招呼的事情，说："哥几个看样子还没喝高，这里再喝点，啤酒免费随便喝，咱们今天几个哥们儿会会，也别讲究地方了，就是啤酒差点，哈啤，口感还可以。"周秘书和孙大夫酒量还可以，我今天已经算是超水平发挥了，所以酒喝得有点费劲，好在老洪酒德不错，又有孙大夫带头冲锋陷阵，一场酒下来，我又喝了差不多两瓶哈啤。要在以往，喝上 3 瓶啤酒，我也就只有找个地方睡觉的份了，今天加在一起，喝了差不多 6 瓶多，居然还没倒下，算是个奇迹了。看样子喝酒也要看场合和心情，老朋友见面，心情愉快，连酒量都长了。

已经下半夜 3 点多了，应该好好睡个觉了，下周将是十分艰苦的一周。

优化小技巧　系统级优化

系统级优化是相对于应用级优化的另一种优化方式。系统级优化是在尽量不修改应用的基础上实现对数据库的优化，一般通过以下手段来实现：

- 调整操作系统参数；
- 调整数据库参数；
- 调整存储；
- 调整网络；
- 进行 I/O 负载均衡处理；

……

系统级优化往往是针对应用的情况调整系统资源配置，使应用系统能够在一种优化的资源配置下运行，从而达到优化的目的。由于不需要深入了解应用系统的细节，因此系统级优化的实施相对比较容易。也正是因为系统级优化不需要了解应用系统的细节，所以系统级优化的成功率相对较低。如果在进行系统级优化时，能够更多地了解应用系统，就可以提高系统级优化的成功率。

系统级优化的重点是找到系统的瓶颈。使用 STATSPACK 报告是检查系统瓶颈的重要手段，因此进行系统级优化可以按照如下步骤进行系统分析：

(1) 进行 STATSPACK 分析，也可以辅以 OEM 的系统健康性报告；

(2) 通过 STATSPACK 报告找到目前系统中超出常规标准的所有情况；

(3) 分析这些情况，并通过辅助检查，找出系统的主要瓶颈；

(4) 根据分析结果，确定系统参数的调整方案。

由于系统级优化方式优化成本相对较小，所以当数据库出现性能问题时，如果应用软件修改的成本比较高，而且进行系统级优化就能够达到目的，那么系统级优化应该是首选方案。

在系统级优化时很容易形成的误区是片面强调某一个方面的优化。以前有个 DBA 问过我，有什么办法能快速让 CPU 使用率下降。我说有个最简单的方法，那就是让 I/O 的性能降低。CPU、内存、I/O 资源是一个铁三角，互相之间会产生互动。做系统级优化，如果把 I/O 优化到极致是不是就是最好的呢？在某些情况下，可能不是，比如在一个 CPU 资源十分紧张的系统下，提升 I/O 性能可能会造成更多的 CPU 开销。另外，由于我们的系统不是一个封闭的应用，或多或少存在一定的排队效应，一旦应用的性能有一点点提升，就可能导致更多的事务涌入系统，导致系统负载上升，CPU 的资源消耗更为严重，当 CPU 出现了瓶颈的时候，可能优化后的系统整体性能不会提升，反而会下降。因此，在做系统级优化时，一定要注意，CPU、内存、I/O 三个资源都不应该出现短板，否则就需要通过调整来达到平衡。也就是一些 DBA 经常说的用 CPU 换 I/O，用内存换 I/O 等。

用内存换 I/O 其实很简单。如果内存空闲较大，而 I/O 压力很大，通过加大 DB Cache 等就可以减少 I/O 的开销。不过，加大 DB Cache 在增加内存使用、减少 I/O 开销的同时，也会增加 CPU 的使用。通过上面的例子，大家需要记住一个原则，就是系统优化的要点是平衡。

优化小技巧　转储文件内容 2：转储对象的段头

在 Oracle 中，每个段都包含一个段头。段头一般来说是某个段第一个扩展的第一个块，包含段的信息（段的扩展情况、FREE LIST 的数量等）。要转储段头的信息，首先需要定位某个段的段头所在的位置。通过下面的查询可以查看到段头的相关信息：

```
SELECT SEGMENT_NAME, HEADER_FILE, HEADER_BLOCK
FROM DBA_SEGMENTS
WHERE OWNER = '&lt;SEGMENT OWNER&gt;' AND SEGMENT_NAME = '&lt;SEGMENT NAME&gt;';
```

比如：

```
        SELECT SEGMENT_NAME, HEADER_FILE, HEADER_BLOCK
        FROM DBA_SEGMENTS
        WHERE OWNER = 'SCOTT' AND SEGMENT_NAME = 'EMP';

SEGMENT_NAME          HEADER_FILE HEADER_BLOCK
--------------------------------------------  --------  ---------------
EMP                            8          7411
```

这个例子说明 EMP 的段头在 8 号文件的 7411 数据块。使用下面的命令，可以把段头转储出来；

```
ALTER SYSTEM DUMP DATAFILE 8 BLOCK 7411;
```

产生的 Trace 文件会存放在 USER_DUMP_DEST 参数指定的位置，如果没有设置这个参数，那么默认的位置是$Oracle_BASE/admin/<sid>/udump。 如果要转储回退段，那么可以这样操作：

```
        SELECT SEGMENT_NAME, HEADER_FILE, HEADER_BLOCK
        FROM DBA_SEGMENTS
        WHERE SEGMENT_NAME = 'RBS01';


SEGMENT_NAME         HEADER_FILE HEADER_BLOCK
-------------------- ----------- ------------
RBS01                          2          514


  alter system dump datafile 2 block 514;
```

或者直接用下面的命令转储：

```
ALTER SYSTEM DUMP UNDO HEADER RBS01;
```

5月15日

业务支撑系统出问题了

今天是正式进入工作状态的一周的开始，上一周事务性的东西太多，因此只是对本次优化工作形成了一个初步的想法。本周的工作很多，除了要完成《系统性能评估报告》外，还要完成《系统性能分析报告》和《系统性能评估指标手册》，这是我们这个项目向客户提交的第一部分技术性报告。报告的内容可能会作为评价我们本次优化工作的基础，因此需要十分细致。我和老于决定分工完成，我负责完成《系统性能分析报告》。

我的计划是按部就班一步一步来，先进行性能分析，确定优化的最佳路径，然后提出正式的优化方案，优化方案必须经过评审，评审后的优化方案需要甲方、集成商和我们三方签字，然后三方根据优化方案共同制订实施计划。这也是我一贯的工作方式，但是实际上很多项目根本无法按照预先设想的完整流程进行，就像这个项目。对于这个病入膏肓的系统，有时候就像救火一样，只能是该出手时就出手了。不过从整个项目的优化效果来考虑，我还是希望这种救火的事情能够越迟发生越好。

今天下午突然爆发了 BSS 系统的性能问题，最初我们认为性能问题最严重的是销账系统，BSS 系统各方面指标还可以，这个突发事件完全颠覆了我的初步印象。今天下午开发商的应用维护人员小孙突然接到很多营业厅的申告，说 BSS 系统性能十分差，营业前台出现了大面积的排队，希望我们能够尽快协助处理一下。对于 Oracle 9i 的数据库，我十分喜欢使用 OEM Performance Manager 进行实时监控，并在每套数据库上安装 STATSPACK，启动了 30 分钟自动采样。在做系统性能优化时，我一般都会要求客户开启自动 STATSPACK 采样，有条件的就开启 30 分钟采样，否则采用 1 小时自动采样。如果采用 30 分钟间隔的 STATSPACK 自动采样，一旦发现系统存在异常，马上做一个采样点，基本上就可以抓住系统存在的主要问题。抓系统的问题，10～15 分钟的 STATSPACK 报告是十分有效的。如果开了 30 分钟自动采样，那么我们只要临时生成一个采样点，就能够获得一份故障发生期间少于 15 分钟采样周期的报告。

当小孙告诉我们 BSS 系统存在严重问题时，我们马上通过 OEM Performance Manager 的系统健康性图表发现系统的负载比较高。根据生成的 STATSPACK 报告，将其和昨天的结果进行比对，发现今天的系统负载比平时高出 30%左右。通过 OSWATCH 的监控数据发现，CPU 或 I/O 都出现了较为严重的问题。

我让小孙找了一个营业厅，咨询一下今天业务量比平时高出这么多，是不是有什么特殊业务。营业厅那边的反馈是没有听说有什么特殊业务，事实上，这是一个错误的信息，由于这个错误信

息，让我们走了很多弯路,业务量有所增长这个信息直接被我们忽略了，我们把分析的重点放在了查找是否存在某些 SQL 走错了执行计划以及是不是碰到了什么 bug 上。经过一个多小时的分析，我们并没有发现核心业务的 SQL 出现执行计划错误，警告日志中也没有发现明显的问题，于是我再次确定肯定是存在什么特殊业务，否则不可能出现这种情况。能够确定这一点是因为昨天我们也采集了 STATSPACK 报告，通过和昨天的 STATSPACK 报告相比，我们发现今天排在前几位的 SQL 是平时在 Top SQL 中看不到的。无论是按照缓冲区获取排序还是按照物理读排序，排在前面的几个 SQL 都占了整个系统的 30%以上，而这几个 SQL 在最近一个月的 STATSPACK 报告中都看不到。通过这样比对，我和老于都觉得这些 SQL 肯定是今天才冒出来的，今天肯定有什么特殊的新业务，或者某种业务今天特别忙。再次和客户确认后，终于找到了答案，这几天的刮刮卡业务十分火爆，业务量差不多是平时的 5 倍以上。听到这个消息，我立即让小孙把这几个 SQL 发给负责刮刮卡业务的开发人员，让他们看看这几个 SQL 是不是刮刮卡业务的。小孙通过 BQQ 把 SQL 发过去，那边立即很肯定地说这几个 SQL 就是刮刮卡业务的。

我们研究了一下这几个 SQL,发现其中几个 SQL 都是对一张 300 万记录的表做全表扫描，而从 SQL 来看，每次返回的记录只有几条，这种情况下，完全是可以通过索引来访问的。添加了一个复合索引后，这几个 Top SQL 的性能有了明显改善，SQL 的成本下降了 99%，过了几分钟，CPU 也出现了久违的 IDLE，系统性能得到了明显的改善。

在调整索引的过程中，我发现这几张相关的表上有很多索引，但都是单列索引，由于这些 SQL 的 WHERE 条件涉及多个列，因此很多 SQL 都是通过几个单列索引扫描，然后取交集，这样的索引效率就相对较低。

经过几层协调，我终于找到了开发这个模块的开发人员，我询问他为什么要设计这样的索引，而不使用复合索引。开发人员的回答让我感到震惊，他说："我不知道什么叫复合索引，我为每个字段都建了索引，Oracle 就应该能用到索引了。"从他的回答上我感到他根本不理解索引的概念，甚至很可能这个项目的整个开发团队都对索引知之甚少，因此他们设计出来的索引可能都存在很严重的问题，今天暴露出来的问题只是冰山一角。

我和老肖讨论了这个问题，决定马上对 BSS 系统的索引进行监控，看看这个系统中的近 2000 个索引，到底哪些用到了，哪些根本没有用到。在下一步的 SQL 优化工作中，索引优化将成为一个重点。老肖主动承担了这个任务，对 BSS 系统的所有常用表进行一次索引分析。方法是使用 STATSPACK 和 SQLA 将主要的 SQL 语句根据缓冲区获取和物理读两种情况进行排序，对排在前 30 位的所有 SQL 进行检查，看看执行计划是否异常，如果发现存在明显索引扫描不合理的 SQL，就将它列出来，交给我进行进一步分析。为了配合 SQL 分析，我建议吧 STATSPACK 的 Snap Level 设置为 6 级，这样在 STATSPACK 采集的数据中就会包含 SQL 的执行计划，可以方便我们对 SQL 进行分析。

我和老肖把 STATSPACK Snap Level 调整到 6 级的想法和老于商量了一下，老于怕提高 Snap Level 会增加系统开销，加重 CPU 的压力。我觉得每 30 分钟采集一次 Snap，目前每次采集 Snap 大概是 20s～30s，如果 Snap Level 提高到 6 级，估计也会在 1 分钟左右完成，对系统的影响不会

太大。其实 BSS 系统在平时的 CPU 负载并不是特别高，只是今天爆发比较明显，只要我们在 25 日业务高峰前把 Snap Level 调整回来，应该对系统不会有多少影响。三个人一讨论，老于觉得我说的也在理，就同意了我的想法。

回到办公室，老肖通过执行一次 STATSPACK Snap 采样把 Snap Level 设置为 6：

```
Sql>exec statspack.Snap(i_snap_level=>6,i_modify_parameter=>'TRUE');
```

为了方便 Top SQL 的提取，我建议老肖修改 STATSPACK 的相关脚本。把?/rdbms/admin/sp-repins.sql 中的 `define top_n_sql = 5` 修改为 `define top_n_sql = 50`，这样我们就可以获得 Top 50 的 SQL 了。

我和老肖正在讨论，看见小孙背着电脑走了过来："老白，还不下班，都 6 点多了。"不知不觉一个下午又过去了，由于下午这个故障，今天该做的事情还有好多没做完，只能带回酒店加班了。我和老肖整理好电脑，发现老于不在办公室里，老肖说老于肯定在门口抽烟呢。出门一看，老于一个人确实坐在那里喷云吐雾呢。我走过去说："老于该下班了，走吧。"老于好像被我惊了一下，抬起头看见我和老肖，做了一个坐下的手势。

我和老肖找了两把椅子坐下来，老于才慢悠悠地说："刚才孙主任又来电话问我们能不能调整一下策略，先搞几个动作，让他们减轻点压力。"

我一听这个马上就急了："老于，你可千万别答应啊，现在我们两眼一抹黑，这个时候做调整不是找死吗？当年海尔的系统那个样子了，我也没有立即动手，起码也是分析了一个多星期才做的第一次调整。"

老于说："老白，你先别急，我刚才也没敢答应，因为这是优化项目的大忌。不过今天下午南塔的一个营业厅由于系统比较慢，有个顾客和营业员产生了争执，把营业员都搞哭了。现在孙主任他们压力很大，我都不知道该怎么拒绝他。"

看样子老于有点心动了，我抽出一支烟点上，猛抽了几口，想让自己冷静一下。这个口子可千万别开，否则我们的优化计划就全泡汤了。

三个人讨论了半天，也没有讨论出个子丑寅卯来。最后我和老于说："算了，这件事我们也别表态了，就拖着吧，能拖到什么时候就到什么时候，实在不行，第一次调整的时间稍微提前一点，但是现在绝对不是动手的好时机。"

今日点评

今天 BSS 系统出现的故障是十分典型的，故障排查的过程在今天的日记中写得相当详细。期间我们犯了一个错误，我们问小孙今天有没有什么特殊业务，他说没有。其实我们不应该这么轻信别人的话，而应该更加相信我们所看到的，从 STATSPACK 报告上我们可以明显看到排在前几位的那几个 SQL 是造成今天问题的元凶，那么我们应该马上把这几个 SQL 交给小孙，让他确认这几个 SQL 是哪个模块产生的，另外，应该马上检查一下这几个 SQL 的执行计划。如果做这些检查，我们很快就可以发现，这几个 SQL 走了全表扫描，那么想办法通过索引去优化就

可以了。

很多 DBA 更为重视技术，而往往忽略了方法。其实在很多优化项目里，方法的正确性远比技术的正确性更为重要。正确的方法可以保证在分析过程中少走弯路。今天的案例如果最初就采用了正确的方法，就可能提前解决问题，不至于浪费这么多时间了。

优化小技巧 索引监控

从 Oracle 9i 开始，可以监控 Oracle 索引的使用情况，具体方法如下：

```
ALTER INDEX <schema>.<index> MONITORING USAGE;
```

对某个索引开启监控后，就可以观察该索引是否被使用：

```
SELECT index_name,monitoring,used,start_monitoring,end_monitoring
FROM v$object_usage;
INDEX_NAME MONITORING USED START_MONITORING END_MONITORING
-------------------------------------------------------------------
AA NO YES 06/04/2006 12:02:38 06/05/2006 13:47:39
AA1 NO YES 06/04/2006 12:02:40 06/05/2006 13:47:39
```

要注意的是，由于 v$object_usage 视图限制了只显示当前用户下被监控的索引的情况，所以通过其他用户登录数据库将无法看到。如果要查看所有用户下被监控的索引的情况，可使用如下 SQL：

```
SELECT u.name owner, io.name index_name, t.name table_name,
decode(bitand(i.flags, 65536), 0, 'NO', 'YES') monitoring,
decode(bitand(ou.flags, 1), 0, 'NO', 'YES') used,
ou.start_monitoring start_monitoring,
ou.end_monitoring end_monitoring
FROM sys.user$ u, sys.obj$ io, sys.obj$ t, sys.ind$ i, sys.object_usage ou
WHERE i.obj# = ou.obj#
AND io.obj# = ou.obj#
AND t.obj# = i.bo#
AND u.user# = io.owner#
```

如果要取消对索引使用情况的监控，可使用下列 SQL：

```
ALTER index <schema>.<index> NOMONITORING USAGE;
```

要注意的是，索引使用情况监控会增加部分系统开销，不过这部分开销基本上可以忽略不计，只是分析结束后一定要注意关闭监控。

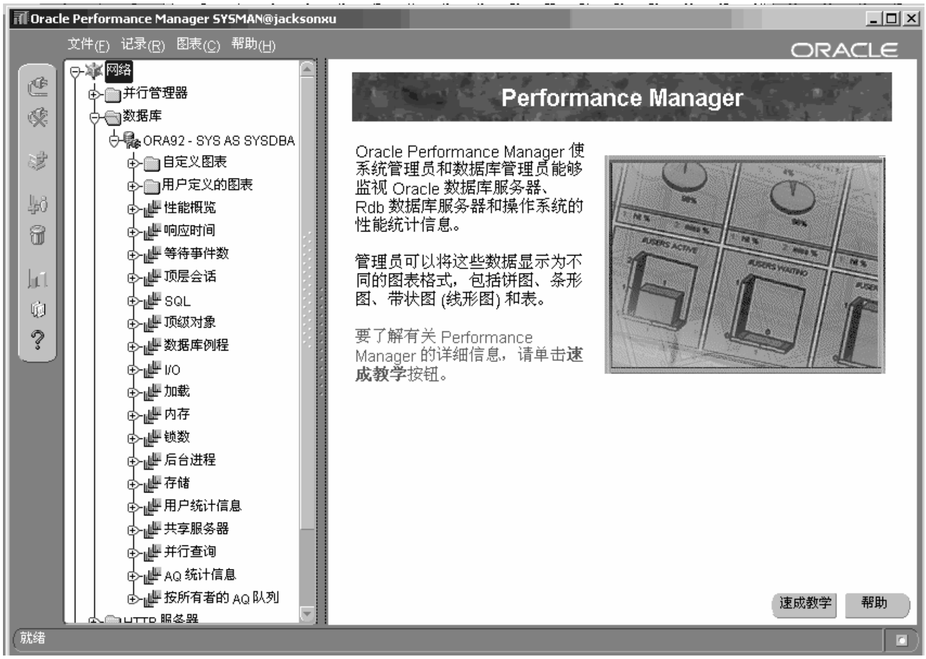
另外要注意的是，v$object_usage 视图是和登录用户关联的，从这里只能看到本用户相关的索引的使用情况，如果通过 sys 登录进去，那么无法看到其他模式下索引的使用情况。

优化小技巧 性能优化时常用的工具

俗话说，工欲善其事，必先利其器。优化工具对优化项目来说也是十分关键的。很多 DBA 都在寻找好的第三方工具，实际上 Oracle 自带的工具也十分有效。在做优化项目时，我们喜欢使用的三件套 OEM、STATSPACK、OSWATCH 都是 Oracle 的工具。对于 Oracle 10g，可以使用 EM、

ADDM、AWR/ASH 和 OSWATCH。

Oracle 9i 的 OEM 是一个十分棒的工具，可能很多 DBA 都用过 OEM，但有些人用过之后可能觉得 OEM 没什么特别的，这是因为很少有人认真用过。OEM 9i 的 Performance Manager、SQL Analyzer 等工具都十分优秀。Performance Manager 中包含了上百个图表，通过这些图表，可以很清晰地观察到数据库的各种状态。



一般来说，启动性能概览图表，就可以看到整个数据库的基本状态。

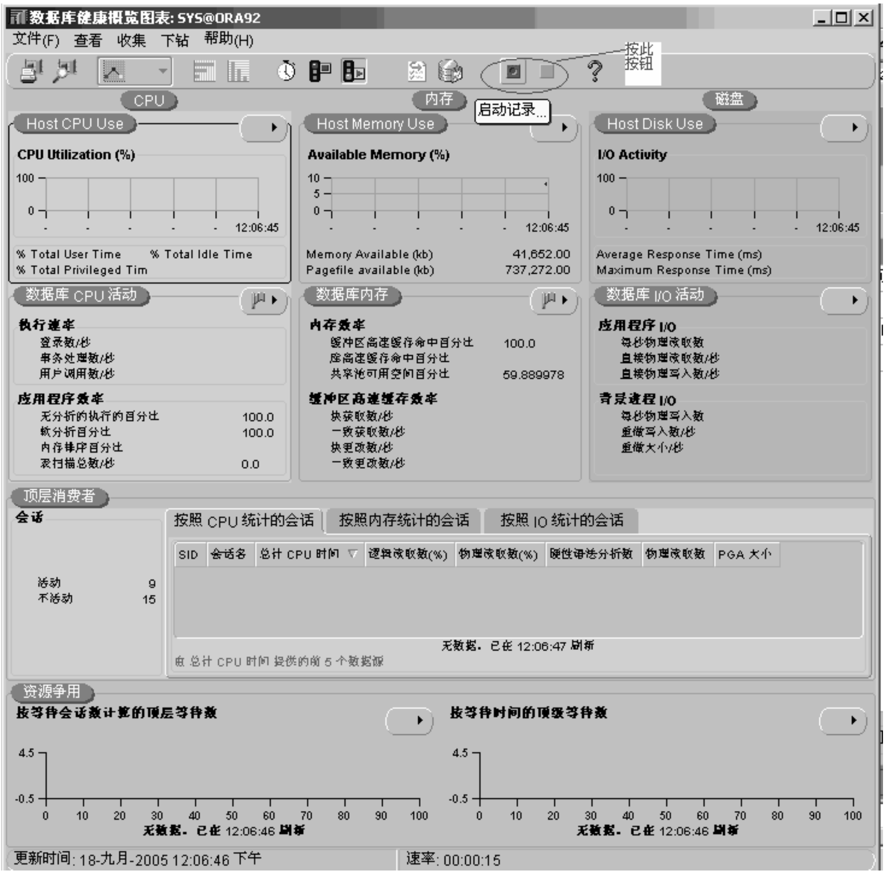
在性能概览图表中，最上面的是 OS 相关情况，包括 CPU 使用率、内存使用情况和 I/O 情况。中间部分包含数据库 CPU 活动、数据库内存、数据库 I/O 活动三个部分的监控内容，包含了 SGA 各个缓冲的情况，redo、dbwriter 等的情况。再下面是会话情况，可以查看到各种视角下的 Top 会话。最底下是系统等待事件。

在 OEM 中，有一个十分好的功能是下钻分析功能，如下图所示。

在很多地方可以通过右键选择“下钻”，进行深入的分析。比如我们发现当时系统等待 db file scattered read 等待比较多，那么可以通过下钻查看哪些 SQL 引起了这个等待，哪些会话引起了这个等待。

Performance Manager 还有一个不错的功能，那就是录像功能。录像可以回放，这个功能是十分有用的。对于 Oracle 10g 的数据库，我们可以使用 ASH 来实现类似的功能，但是对于没有 ASH 功能的 Oracle 9i 数据库来说，这个功能可以帮助 DBA 解决大问题。有一次一个客户打电话给我说系统性能有问题，赶到现场需要 30 分钟，而这个问题出现的时间大约是 5～10 分钟，所以比

较难捕捉。我接到电话后让他们做了两件事情：一是马上做一个 STATSPACK 的 Snap，二是启动 OEM 的几个监控视图，并且进行录像。我到达现场后，根据 OEM 的录像回放，很快定位到主要的等待事件的情况，并且从 STATSPACK 报告中得到了印证。这个困扰了客户很久的问题，在这两个工具的帮助下很容易就定位了。



OSWatch（简称 OSW）是 Oracle 的一个操作系统监控工具，可以从 Metalink 上下载（Node 301137.1）。OSW 是一个轻量级的操作系统监控工具，通过一个后台进程进行调度，使用 `vmstat`、`iostat` 等操作系统命令对操作系统的主要指标进行监控，并形成历史数据。OSW 对系统的开销十分小，部署也十分方便，只要用 tar 解开安装包就完成安装了。OSW 采集的信息可以通过 OSWG 工具生成图表。OSWG 是一个 Java 的应用，可以在服务器上运行。如果服务器支持图形界面，可以直接在服务器上观看，如果服务器不支持图形界面，也可以直接生成图形文件，然后下载到微机上查看。



优化小技巧　转储文件内容 3：转储控制文件

通过设置系统事件，可以把控制文件转储到 Trace 文件：

```
ALTER SESSION SET EVENTS 'immediate trace name controlf level <level>';
```

<level>的取值如下表所示。

Level	转储的内容
1	文件头信息
2	文件头信息,数据库信息记录和检查点进程记录
3	所有的记录,仅仅包含最新和最老的循环使用类型的记录
4	类似3，不过包含4条最新的循环使用类型的记录
5+	类似3，循环使用类型的记录是每个级别的双倍

5月18日 Top SQL

通过这几天的分析，这个系统的主要问题基本上清晰了。由于受到硬件资源的限制，这个系统一直处于硬件资源的临界状态运行，一旦某个业务出现了 10%以上的增长，就会导致 CPU 或者 I/O 出现瓶颈，从而导致整个系统性能大幅度下降。因此如果某个人通过 SQL*Plus 上去跑一个查询，就可能导致系统在半个小时里出现严重的性能问题。虽然运维部已经三令五申，从早上 8：30 到晚上 18：00 之间严禁通过 SQL*Plus 连上去，不过由于业务需要，假如领导临时要一个数据，而这个数据又是系统无法提供的，那么就必须连上去查询了。

从目前的角度来看，扩容硬件是最好的解决方案，但是由于甲方决定采用大集中的方案，各省的系统将在 2 年后全部纳入大区进行管理，所以目前硬件投资和扩容是不可能的，这其实也是集团公司领导决定进行调优的初衷。

如果排除了硬件扩容的方案，那么就不可避免地要进行应用的优化。而以目前应用开发商的情况来说，如果涉及 SQL 的优化，那就是个很大的问题。这个项目要求的周期这么紧张，如果在 6 月 20 日前，应用厂商无法将我们提出要修改的 Top SQL 修改完毕，那么优化的效果就无法在 7 月份评估完成，计划中的 7 月中旬进行的验收也就无从谈起了。7 月上旬完成整个项目的验收是小齐在确定这个项目进度时一再强调的，因为整个北方公司 9 个省的调优工作必须在今年 12 月底前完成，而能否拿下北方公司全部的合同，必须看辽宁的优化效果，如果优化效果不理想，那么北方公司可能会终止这个项目。如果 7 月上旬无法完成验收，那么验收可能就要推到 8 月份进行，验收后安排的那次研讨会的时间只能推迟到 9 月份，研讨会后再启动整个北方公司的项目，到签署合同，估计就要到 12 月了，这样，12 月底前完成北方公司整个项目实施就绝无可能了。而这个项目是本年度的项目，跨财务年度是不现实的。为了保证其他各省的项目能够如期在 11 月份开始启动，我们这个项目必须控制在 7 月上旬验收。

我刚刚接到这个项目需求时，对 7 月份验收的信心还是很足的，从初步拿到的资料分析，这个系统提升 30%不难，甚至不需要优化 SQL 就可以做到。而从目前的情况来看，这个系统需要的不仅仅是 30%的性能提升而已。况且对于这种处于容量临界状态运行的系统，其排队效应是十分大的，通过一定手段的优化，系统中的等待事件可以大幅度下降，甚至可以下降 50%。不过这种下降对整体性能的提升作用并不大，因为系统中还积压了大量等待处理的事务，一旦通过调整后，系统的等待事件下降了，那么大量的正在排队的事务会挤入，这样刚刚释放出来的资源又会被这些排队的事务占用，优化后降低的资源消耗仍然无法消化掉这些排队的事务，这样系统的整

体性能并不会有明显的改善。从这方面来讲，如果不做 SQL 优化，通过其他应急的优化，可以短期解决一定的性能问题，甚至可能做到性能提升 30%的目标，完成验收，但是随着业务量的增长，几个月后，系统资源瓶颈将会再次出现，这个系统可能又会回到优化前的状态。

基于上面的考虑，SQL 优化是必需的，而且必须做到位，这样这个项目才可能顺利达成目标，在大集中前维持稳定的运行状态。到目前为止，已经发现对系统性能影响较大的 Top SQL 30 余个，这些 SQL 消耗了超过 60%的 CPU 资源和 50%以上的 I/O 资源，如果能够优化，就可以大幅度减少资源开销，实现我们的目标。不过这些 SQL 提交到开发商手中已经有几天了，没有任何反馈信息。我自己也做过几年开发工作，知道软件开发人员的难处，特别是现在这种情况，一套系统全国几十个地方在用，要修改任何一处，都会涉及版本管理的问题，想要从他们那里很快获得反馈确实有点难度。

下午收到一个邮件，是开发商关于 SQL 优化的反馈，感到有些意外，因为邮件的标题是“SQL 优化已完成，将于 5 月 23 日前完成实施”。在两三天内完成了 30 多个 Top SQL 的优化，其中不乏文本长度超过 20KB 的超大 SQL，这么高的效率是前所未有的。打开附件，我顿时有一种被愚弄的感觉。原来开发厂商的处理方案中，除了几个明显缺少索引的 SQL 通过添加索引来解决外，其他的 SQL 全部被说明为“已经经过两年多的优化，已无优化余地”，实际上应用开发厂商几乎拒绝了所有 SQL 的修改。

老于不停地抽着烟，我们两个在机房外的楼梯间里研究了个把小时，也没有什么好的对策。从这个邮件上来看，我们可能要面临得不到应用开发厂商支持独立作战的不利局面，要想达成优化目标，难度会大大增加。我们必须独立完成 SQL 的优化，把优化方法以及结果提交给客户，然后由客户去压应用厂商修改软件。这样的话我们的优化建议必须在 6 月 10 日之前提交给客户，否则应用开发厂商可以以“时间太短，无法完成”为由再次拒绝我们的方案。从现在来看，我们只有不到一个月的时间，要完成 50～70 个 SQL 的优化工作，平均每天要完成 1～2 个 SQL，其中大多数 SQL 都是超过 5 个表连接的超大型 SQL，工作量之大，前所未有。

为了减少这段时间由于突发性故障引起的问题，我和老于向孙主任提出建议，这段时间一概禁止客户端通过 SQL*Plus、PL/SQL Developer 等工具连到数据库上做大的查询，一旦发现这种操作，我们会主动杀掉这些进程。孙主任同意了这个提议，决定今天就发文给各个部门。我这边也安排开发商的小孙负责这件事情。我给了小孙一个查找大开销 SQL 的脚本，让他定期跑一下这个脚本，一旦发现某个不是应用程序进行的大型操作，就直接杀掉会话。这个脚本是 Metalink 上的一个脚本，内容如下：

```
SELECT
L.sid ssid,
substr(OPNAME,1,15) opname,
target,
trunc((sofar/totalwork)*100) pct,
to_char(60*sofar*8192/(24*60*(last_update_time - start_time))/1024/1024/60, '9999.0')
Rate,
elapsed_seconds/60 es,
```

```
time_remaining/60 tr,PROGRAM,MACHINE
FROM v$session_longops L,V$SESSION S
WHERE time_remaining > 0 AND L.SID=S.SID
ORDER BY start_time
/
                                                          I/O    Time Time
  SID OPNAME      TARGET                              PCT Rate/m  Ran Left PROGRAM       MACHINE
----- ---------- ------------------------- ---- ---- ---- ---- ----- -------------
 3086 Table Scan  CRTA_RPT.RPT_TRANSFER_BAR     36.00  478.  .3 .58 oracle@mrp1 (J0 ods2
 2668 Table Scan  MONITOR_DB.FACT_REAL_DELAY_D 37.00  386.  .2 .37 oracle@monitord monitordb
```

如果 PROGRAM 是 SQL*Plus 或者其他第三方工具，在做业务表的大型扫描操作时，这种操作可能会引起系统性能问题。通过 MACHINE 可以知道是谁在做这个操作，除了几个特殊的 MACHINE 外，所有的做这种危险操作的会话都在立即清理的范畴之内。我们授权小孙，可以随时杀掉这些会话。

对于这样的系统，禁止由于某个人在业务高峰期间执行一些高开销 SQL，可以有效防止一些不必要的系统故障。这个方法在很多以往的优化项目中都采用过，效果还不错，因此这一次我们在沈阳也使用了。具体能有多大效果，就看小孙是不是尽心尽责了，因为实践证明，光靠行政命令是无法杜绝这样的操作的，孙主任的通知虽然发出去了，不过又有几个人能够遵守呢？实际上，发出这个通知只是为小孙杀会话找一个合适的理由。

为了解决另一项目出现的紧急情况，我将在本周末暂时离开这个项目一段时间，因此必须让老熊马上结束南京的事情，立即加入这个项目，否则时间就来不及了。下午和老熊通了电话，他最快这个周末可以离开南京。老于听到消息也松了一口气，老熊的提前加入，是十分关键的，在 SQL 优化方面，老熊的经验要比老于和老肖丰富得多，他可以填补我离开的空缺，确保项目的进度。

优化小技巧 查找 Top SQL

发现 Top SQL 的方法很多，最常用的方法是从 STATSPACK 报告中查找 Top SQL。另外，OEM Top SQL 工具 OEM SQL Analyzer、10g ADDM 等都是查找 Top SQL 的好工具，还可以从 v$sqlarea 中直接查询。

查找 Top SQL 的依据包括以下几个方面。

- 查找总的缓冲区获取比较高的 SQL，并根据平均每次执行缓冲区获取的数量判断优化的余地有多大。优化这些 SQL，有助于减少 CPU 开销以及数据缓冲池相关的闩锁竞争。
- 查找总的物理读比较高的 SQL，并根据平均每次执行物理读的数量判断优化的余地有多大。优化这些 SQL，有助于减少 I/O 开销和 CPU 开销。
- 单次执行开销较大的 SQL 也属于重点优化之列，但是对于某些执行次数不多、总的开销排名不靠前的 SQL，优化后对系统的影响也许并不十分大。由于 SQL 优化是个十分消耗人力资源的工作，所以有时候这类 SQL 不一定作为优化的重点。

查找 Top SQL 的主要工具包括以下几种。

- STATSPACK 报告：STATSPACK 报告中的 Top SQL 是比较全面的，而且也具有一定的代表性。不过 STATSPACK 报告中的 SQL 一般来说是残缺不全的。可以通过 sprepsql.sql 脚本生成 SQL 报告，这份 SQL 报告里包含完整的 SQL 文本和 SQL 执行的开销，如果 STATSPACK 的 Snap Level 是 6 级或者以上，报告里还将包括 SQL 的执行计划等信息。
- AWR 报告：和 STATSPACK 报告类似，Oracle 10g AWR 也可以查找 Top SQL。AWR 的基本 Snap 级别就包含了 SQL 的执行计划。因此，可以使用 awrsqrpt.sql 直接生成某个 SQL 的 AWR 报告。
- 9i OEM Top SQL： 9i OEM 里有一个 Top SQL 工具，可以根据不同的过滤条件和排序顺序查找各类 Top SQL。
- 9i SQL Analyzer 优化工具：SQL Analyzer 也是 OEM 附带的一个工具包，该工具包是一个很不错的 SQL 优化工具，不仅可以查找 Top SQL，还可以辅助分析 SQL，并可以提出一些优化建议。
- 10g ADDM：ADDM 提供的 SQL 优化功能更为强大，优化建议功能也更强。ADDM 可以根据 SQL 对系统的影响度，对 Top SQL 进行排序。找到 Top SQL 后，可以立即使用优化向导进行优化分析。
- TRCA：Oracle 分析 10046 Trace 文件的一个工具，可以从 10046 Trace 文件中分析 Top SQL。TRCA 可以从 Metalink 上下载（文档 224270.1）。
- SQLAREAT：一个 Top SQL 分析工具，从 SQL area 中抓取 Top SQL。SQLAREAT 可以从 Metalink 上下载（文档 238684.1）。

在上面介绍的工具中，STATSPACK、AWR、ADDM 是根据某个时间段中 SQL 对整个系统的开销来进行分析和排序的，而其他工具侧重于对当前系统中累计信息进行排序，因此通过不同的工具查找出来的 Top SQL 可能不同。在做优化时，不能仅仅考虑单条 SQL 的开销，更要考虑那些执行次数比较多、总开销较大的 SQL。

优化小技巧　索引设计的一些建议

索引对改善数据库访问性能的作用十分巨大，设计合理的索引对于系统性能调整至关重要，而使用索引又是数据库开发过程中最困难的一点。在设计索引时要注意以下几个方面。

- 在 OLTP 系统中，尽量避免全表扫描，尽量使绝大多数操作都通过索引访问。
- 数据量很大并且经常变动的表上的索引不易过多，过量的索引会导致插入、更新和删除操作变慢，产生大量的 I/O。
- 如果索引数量过多，并且很多列包含在多个索引中，就要考虑使用合适的复合索引来替代几个其他的索引。
- 小表不要建立索引，通过索引访问速度可能更慢，把小表放入 keep 池可能效率更高。
- 索引的 `PCTFREE`、`INITTRANS`、`MAXTRANS` 参数设置十分重要，特别是对于变化十分大的索引。

- 对于比较大的索引，使用索引分区会改善效率。
- 位图索引对于列表类值的效果较好（索引也比较小），但是位图索引不适合变化十分频繁的表，因为位图索引在变更时会锁住相应的一组记录而不仅仅是一条记录。
- 在某些情况下，使用函数索引可以避免大量不必要的全表扫描。
- 如果索引包含了查询需要的所有信息，查询就不需要访问表的数据，这可以大大提高访问效率。因此，要注意建立合理的复合索引，适当添加一些额外的字段可能会起到很好的优化效果，另外也要注意 SQL 语句的合理性，绝对不检索不适用的列。
- 用好索引组织表。索引组织表是一种十分特殊的表，这个表的结构本身就是一个索引。
- 可以使用反转索键值索引来消除索引相关的热块。
- 随着数据的变化，索引的效率会下降，因此定期重建索引对于性能提升有很大帮助。
- 使用 CBO 优化器的情况下，保证表和索引的数据得到了良好的分析是保证优化器选择最佳执行计划的关键。
- 删除所有不必要的索引。Oracle 9i 提供的新特性可以使 DBA 能够跟踪索引的使用情况，用这个功能找到不使用的索引，并把这些索引删除掉。
- 如果某个索引确实可以提高 SQL 的性能，减轻系统的整体负载，那么这个索引就是必须加的，哪怕这张表上以前已经创建了 20 个索引。一张表上的索引数不能超过多少这类说法是没有依据的，从实际需求出发才是最关键的。

优化小技巧　什么是索引的集群因子

什么是集群因子（Clustering Factor）呢？集群因子是通过一个索引扫描一张表时需要访问的表的数据块的数量。集群因子的计算方法如下：

(1) 扫描一个索引；

(2) 比较某行的 ROWID 和前一行的 ROWID，如果这两个 ROWID 不属于同一个数据块，那么集群因子增加 1；

(3) 整个索引扫描完毕后，就得到了该索引的集群因子。

以上面集簇因子的计算方式可以看出，集簇因子反映了索引范围扫描可能带来的对整个表访问过程的 IO 开销情况，如果集群因子接近于表存储的块数，说明这张表是按照索引字段的顺序存储的。如果集群因子接近于行的数量，那说明这张表不是按索引字段顺序存储的。在计算索引访问成本时，集群因子十分有用。集群因子乘以选择性参数就是访问索引的开销。

如果这个统计数据不能反映出索引的真实情况，那么可能会造成优化器错误地选择执行计划。另外，如果某张表上的大多数访问是按照某个索引做索引扫描，那么将该表的数据按照索引字段的顺序重新组织，可以提高该表的访问性能。

优化小技巧　转储文件内容 4：转储控制文件

使用 `ALTER SYSTEM DUMP LOGFILE` 可以转储在线日志文件。也可以使用 `ALTER SESSION`

命令去转储重做日志头的信息。在数据库处于 NOMOUNT、MOUNT 或者 OPEN 状态时，就可以使用上面的命令。当操作系统相同的情况下，可以通过客户端去转储另外一个数据库的重做日志。重做日志的信息会被写入一个 Trace 文件。

可以用下面的方法转储在线日志。

- 根据 DBA（Data Block Address）转储记录。
- 根据 RBA（Redo Block Address）转储记录。
- 根据时间点去转储记录。
- 根据层次和操作码转储记录。
- 转储文件头信息
- 转储整个重做日志文件

下面分别介绍一下有关的命令。

1. 根据 DBA 转储在线日志记录

转储跟某个或某些数据块相关的重做日志记录，转储出来的信息只和这些数据块相关，命令如下：

```
ALTER SYSTEM DUMP LOGFILE 'filename'
 DBA MIN fileno . blockno
   DBA MAX fileno . blockno;
比如：
        ALTER SYSTEM DUMP LOGFILE
 'u01/oracle/V7323/dbs/arch1_76.dbf'
        DBA MIN 5 . 31125
        DBA MAX 5 . 31150;
```

这个命令转储 5 号文件的 31125～31150 块相关的记录。

2. 根据 RBA 转储在线日志记录

通过重做地址的范围转储在线日志的记录，语法如下：

```
ALTER SYSTEM DUMP LOGFILE 'filename'
   RBA MIN seqno . blockno
   RBA MAX seqno . blockno;
```

比如说

```
ALTER SYSTEM DUMP LOGFILE 'u01/oracle/V7323/dbs/arch1_76.dbf'
   RBA MIN 2050 . 13255
   RBA MAX 2255 . 15555;
```

3. 根据时间点转储在线日志记录

可以通过时间范围来转储某段时间内产生的重做记录，语法如下：

```
ALTER SYSTEM DUMP LOGFILE 'filename'
   TIME MIN value
   TIME MAX value;
```

比如：

```
ALTER SYSTEM DUMP LOGFILE'u01/oracle/V7323/dbs/arch1_76.dbf'
```

```
TIME MIN 299425687
TIME MAX 299458800;
```

下面介绍一下如何在 redo dump time 和 date 类型之间进行相互转换。

date 转换为 redo dump time 格式的脚本如下（转载 Metalink 上的脚本）：

```
SET ECHO off
REM NAME:    TFSTM2RD.SQL
REM USAGE:"@path/tfstm2rd"
REM ------------------------------------------------------------------------
REM REQUIREMENTS:
REM    None.
REM ------------------------------------------------------------------------
REM AUTHOR:
REM    Anonymous
REM    Copyright 1996, Orqacle Corporation
REM ------------------------------------------------------------------------
REM PURPOSE:
REM    Converts a standard date into redo dump time format.
REM ------------------------------------------------------------------------
REM EXAMPLE:
REM    Enter day (DD/MM/YYYY) ? 08/07/1996
REM    Enter time (HH24:MI:SS) ? 12:05:05
REM
REM    REDO_YEAR REDO_MONTH REDO_DAY REDO_HOUR REDO_MIN REDO_SEC
REM    --------- ---------- -------- --------- -------- --------
REM         1996          7        8        12        5        5
REM
REM    EDO_TIME
REM    ----------
REM     273845105
REM
REM ------------------------------------------------------------------------
REM DISCLAIMER:
REM    This script is provided for educational purposes only. It is NOT
REM    supported by Oracle World Wide Technical Support.
REM    The script has been tested and appears to work as intended.
REM    You should always run new scripts on a test instance initially.
REM ------------------------------------------------------------------------
REM Main text of script follows:

undefine redo_day
undefine redo_hhmiss

accept redo_day prompt "Enter day (DD/MM/YYYY) ? "
accept redo_hhmiss prompt "Enter time (HH24:MI:SS) ? "

column redo_year  new_value redo_year  format 9999
column redo_month new_value redo_month format 9999
column redo_day   new_value redo_day   format 9999
column redo_hour  new_value redo_hour  format 9999
column redo_min   new_value redo_min   format 9999
column redo_sec   new_value redo_sec   format 9999
column redo_time  new_value redo_time
```

```
set verify off

SELECT
  to_number(to_char(to_date('&redo_day &redo_hhmiss',
         'DD/MM/YYYY HH24:MI:SS'),'YYYY')) redo_year,
  to_number(to_char(to_date('&redo_day &redo_hhmiss',
         'DD/MM/YYYY HH24:MI:SS'),'MM'))   redo_month,
  to_number(to_char(to_date('&redo_day &redo_hhmiss',
         'DD/MM/YYYY HH24:MI:SS'),'DD'))   redo_day,
  to_number(to_char(to_date('&redo_day &redo_hhmiss',
         'DD/MM/YYYY HH24:MI:SS'),'HH24')) redo_hour,
  to_number(to_char(to_date('&redo_day &redo_hhmiss',
         'DD/MM/YYYY HH24:MI:SS'),'MI'))   redo_min,
  to_number(to_char(to_date('&redo_day &redo_hhmiss',
         'DD/MM/YYYY HH24:MI:SS'),'SS'))   redo_sec
  FROM dual;

SELECT    (  (  (  (  (        ( &redo_year - 1988 )
                          ) * 12
                               +  ( &redo_month - 1   )
    ) * 31
                               +  ( &redo_day   - 1   )
           ) * 24
                               +  ( &redo_hour        )
          ) * 60
                               +  ( &redo_min         )
           ) * 60
                               +  (  &redo_sec        )       redo_time
  FROM dual;
```

下面是使用该脚本的例子：

```
Enter day (DD/MM/YYYY) ? 08/07/1996
Enter time (HH24:MI:SS) ? 12:05:05

REDO_YEAR REDO_MONTH REDO_DAY REDO_HOUR REDO_MIN REDO_SEC
--------- ---------- -------- --------- -------- --------
     1996          7        8        12        5        5


 REDO_TIME
----------
 273845105
```

将 redo dump time 转为 date 格式的脚本如下（转载 Metalink 上的脚本）：

```
SET ECHO off
REM NAME:    TFSRD2TM.SQL
REM USAGE:"@path/tfsrd2tm"
REM ------------------------------------------------------------------------
REM REQUIREMENTS:
REM None.
REM ------------------------------------------------------------------------
REM AUTHOR:
```

```
REM    Anonymous
REM    Copyright 1996, Oracle Corporation
REM -----------------------------------------------------------------------
REM PURPOSE:
REM    Will convert redo dump time into a readable date.
REM
REM -----------------------------------------------------------------------
REM EXAMPLE:
REM    Enter redo time ? 273845105
REM
REM    REDO_YEAR REDO_MONTH REDO_DAY REDO_HOUR REDO_MIN REDO_SEC
REM    --------- ---------- -------- --------- -------- --------
REM         1996          7        8        12        5    5
REM
REM -----------------------------------------------------------------------
REM DISCLAIMER:
REM    This script is provided for educational purposes only. It is NOT
REM    supported by Oracle World Wide Technical Support.
REM    The script has been tested and appears to work as intended.
REM    You should always run new scripts on a test instance initially.
REM -----------------------------------------------------------------------
REM Main text of script follows:

undefine redo_time

accept redo_time prompt "Enter redo time ? "

column redo_year  new_value redo_year  format 9999
column redo_month new_value redo_month format 9999
column redo_day   new_value redo_day   format 9999
column redo_hour  new_value redo_hour  format 9999
column redo_min   new_value redo_min   format 9999
column redo_sec   new_value redo_sec   format 9999

set verify off

SELECT
trunc(trunc(trunc(trunc(trunc(&redo_time/60)/60)/24)/31)/12)+1988  redo_year,
mod(trunc(trunc(trunc(trunc(&redo_time/60)/60)/24)/31),12  )+1  redo_month,
mod(trunc(trunc(trunc(&redo_time/60)/60)/24),  31)+1       redo_day,
mod(trunc(trunc(&redo_time/60)/60),24)                 redo_hour,
mod(trunc(&redo_time/60),60)                          redo_min,
mod(&redo_time,60)                            redo_sec
FROM dual;
```

执行的例子如下：

```
Enter redo time ? 273845105

REDO_YEAR REDO_MONTH REDO_DAY REDO_HOUR REDO_MIN REDO_SEC
--------- ---------- -------- --------- -------- --------
     1996          7        8        12        5        5
.
```

4. 根据层次和操作代码转储记录

根据层次和操作码可以转储某种类型的记录，命令格式如下：

```
ALTER SYSTEM DUMP LOGFILE 'filename'
   LAYER value
   OPCODE value;
```

比如：

```
ALTER SYSTEM DUMP LOGFILE 'u01/oracle/V7323/dbs/arch1_76.dbf'
LAYER 11
OPCODE 3;
```

5. 转储文件头信息

使用下列命令可以转储文件头信息：

```
ALTER SESSION SET EVENTS'immediate trace name redohdr level 10';
```

6. 转储整个重做日志

使用下列命令可以转储整个重做日志：

```
ALTER SYSTEM DUMP LOGFILE 'filename';
```

比如：

```
ALTER SYSTEM DUMP LOGFILE 'u01/oracle/V7323/dbs/arch1_76.dbf';
```

ALTER SYSTEM DUMP LOGFILE 'filename'命令既可以转储在线日志文件，也可以转储归档日志文件。

对于 Oracle 7 或以前的数据库，Oracle 可以把归档日志直接写入磁带。Oracle 也支持从磁盘或者磁带上读取归档日志。如果从磁盘上读取归档日志，文件名的格式是：

```
D:MINIDISK:DATASET
```

其中 DATASET 是文件名。比如：

```
ALTER SYSTEM DUMP LOGFILE 'd:golddba.230.230.readpass.multpass:l123.ora';
```

5月19日 南京的死锁问题

由于江苏的项目出现一个紧急问题，我只能暂时离开项目组一周的时间，幸亏老熊已经提前加入，给我和老于一个不小的安慰。老熊从事DBA工作差不多10年了，其严谨和执着的态度以及丰富的经验都是我们所需要的。

老熊是早上从南京过来的，而我要飞南京，所以决定在机场见面。我计算过，如果老熊的航班不晚点一个小时以上，我们在机场就有足够的时间做一个交接。还好老熊的航班十分准点，我们两个在机场找了一个茶座，我向老熊介绍了这边的情况，重点介绍了开发商那边的态度以及我们的目标。同时我也把这几天采集的Top SQL交给了老熊，并且告诉他其中有几个SQL是最关键的，不过这些SQL都比较大，大多数是五六张表连接的大型查询，我建议他首先不要碰那几个特别大的查询，先从简单的入手。老熊说等他仔细看看这些SQL后再做决定。和老熊简单交接了文档后，我就匆匆离开了沈阳。

老熊和我一起合作有一两年了，2005年开始他就负责南京那边的业务，江苏的客户对老熊的评价是水平很高，人品也好，干活比较踏实。从这几天我和老于的接触来看，老熊和老于的性格相仿，我相信他们之间的合作不会存在什么问题。老熊的酒量也很不错，没事的时候喜欢喝点小酒，这一点和老于肯定会很投缘。唯一不足的是老熊不抽烟，也从来没有和抽烟的人一起做事，和老于这杆老烟枪在一起工作，估计需要戴一个防毒面具才行。

江苏的问题比较棘手，一个月前，他们的BSS系统出现了由于ORA-60错误而导致整个实例被短时挂住的问题。开始时这个问题只是在某个本地网的系统中出现，后来随着业务量的增加，好几个本地网都出现了类似的问题，有时甚至会导致整个系统被挂起5～10分钟，严重影响了业务。

Oracle公司也派工程师对这个故障做过几次相关的分析。Oracle公司方面初步确定为bug引起，建议他们先打一个补丁。由ORA-60错误而导致实例挂住的bug我也遇到过，所以建议他们先按照Oracle的要求打补丁升级，不过打过补丁之后，故障并没有消失，出现的频率也未发生任何改变。

此前，通过江苏的李工我了解到，故障发生时的现象是数据库被挂住，无法进行任何操作（包括登录）。通过操作系统监控，发现总的CPU使用率并不高，但是总会有一两个CPU的使用率超过90%，甚至接近或达到100%。其他的CPU都很空闲。该故障出现时，都伴随有死锁的报错。

```
ORA-000060: Deadlock detected. More info in file
/oracle/app1/oracle/product/9.2/admin/bsssz/udump/bsssz_ora_29184.trc。
```

在伴随出现该故障的同时，会发现大量的 ORA-600[KKSSCL-INF-IN1-LOOP]错误信息。

```
Tue Apr 18 13:43:31 2006
Errors in file /oracle/app1/oracle/product/9.2/admin/bsssz/udump/bsssz_ora_7554.trc:
ORA-00600: 内部错误代码，参数: [kksscl-inf-inl-loop], [2500], [666], [713], [1427],
[1427], [], []
Tue Apr 18 13:45:30 2006
Errors in file /oracle/app1/oracle/product/9.2/admin/bsssz/udump/bsssz_ora_10892.trc:
ORA-00600: 内部错误代码，参数: [kksscl-inf-inl-loop], [2500], [592], [603], [1211],
[1211], [], []
Tue Apr 18 13:46:23 2006
Errors in file /oracle/app1/oracle/product/9.2/admin/bsssz/udump/bsssz_ora_10892.trc:
ORA-00600: 内部错误代码，参数: [kksscl-inf-inl-loop], [2500], [592], [603], [1211],
[1211], [], []
Tue Apr 18 13:48:25 2006
Errors in file /oracle/app1/oracle/product/9.2/admin/bsssz/udump/bsssz_ora_8253.trc:
ORA-00600: 内部错误代码，参数: [kksscl-inf-inl-loop], [2500], [424], [454], [909], [909],
[], []
Tue Apr 18 13:48:35 2006
Errors in file /oracle/app1/oracle/product/9.2/admin/bsssz/udump/bsssz_ora_11751.trc:
ORA-00600: 内部错误代码，参数: [kksscl-inf-inl-loop], [2500], [424], [454], [909], [909],
[], []
Tue Apr 18 13:49:18 2006
Errors in file /oracle/app1/oracle/product/9.2/admin/bsssz/udump/bsssz_ora_8253.trc:
ORA-00600: 内部错误代码，参数: [kksscl-inf-inl-loop], [2500], [424], [454], [909], [909],
[], []
```

通过 Top 命令发现，有两个数据库服务器进程十分繁忙。如果杀掉繁忙的数据库进程，系统就会恢复正常。有时候，不杀掉进程，系统也会自动恢复正常。经过观察，被杀掉的繁忙进程就是产生死锁的进程。同时系统中存在大量等待 enqueue hash chains 闩锁的会话。

Oracle 公司的技术支持人员认为这是由于 bug 2235386 引起的，并且已经打了 2235386 补丁。打了补丁后，仍然出现了相同的故障，并且故障的频率并没有改变，说明补丁 2235386 并不能彻底解决本故障。bug 2235386 可能只是一个类似的 bug，也是由于死锁检测导致系统被挂住。我最初也怀疑是由该 bug 引起，但是打完补丁后问题仍然存在，说明这个问题不是那么简单，这个案例有可能是 Oracle 尚未解决的 bug。

沈阳飞南京需要两个多小时，在飞机上我认真阅读了 bug 2235386 的资料。这些资料是我这几天从网上下载的，关于这个问题，我也已经研究了个把星期了，除了对 bug 2235386 有了一定的认识外，其他方面还没有什么头绪。由于不知道死锁检测的算法，所以很多方面只能是猜测，通过 parent enqueue hash chains 等待这个情况来看，在死锁检测时可能有会话持有了这个闩锁。这个父闩锁是获取锁资源的核心，一旦这个闩锁被占用，系统中所有涉及锁操作的工作都必须等待，而死锁检测是经常性任务，这么频繁地持有 parent enqueue hash chains 显然对系统的整体性能有很大的影响。Oracle 内部实现死锁检测为什么要持有这个闩锁呢？

从我手头找到的这些资料来看，这个 bug 最早发现是在 8.1.7 上，当时认为 8.1.7.4 已经修复

了这个 bug。不过后来在 9.2.0.1 和 9.2.0.2 上再次被发现，9.2.0.3 里包含了这个补丁的修复，但是在 9.2.0.3 或者更高的版本中又有大量该 bug 导致问题的报告。在一份 bug 报告里，我看到了一些线索。死锁诊断时，需要做进程状态转储，在这之前，会持有 parent+children enqueue hash chains 闩锁。这个闩锁被持有后，其他的锁请求就会被全部冻结，因此如果进程状态转储的持续时间比较长，就可能会使整个系统挂住。

由于事情十分紧急，所以我一下飞机就直接带着行李赶到了客户现场，李工已经根据我们在机场沟通的方案，通过 LogMiner 分析了无锡数据库 5 月 10 日出现故障的时间段前后 1 个小时的日志，发现 26 分 17 秒到 27 分 56 秒之间没有任何 SQL 语句执行，说明这个期间数据库完全挂住（一般情况是发生了内部死锁或者系统内部资源不足造成的等待某资源释放）。在 23 分到 26 分 17 秒之间有一个类似 DDL 操作的迹象，5 月 10 日 20：16：10，产生了大量内部操作和对 SEG$和 TSQ$的操作。从上述信息中还无法确定该操作和系统挂住是否有直接的联系。可以确认的是，该操作和段扩展没有关系。除此之外，在整个分析时段中没有有明显异常的 SQL。另外一个特点是，在 ORA-60 挂起系统时，会伴随出现 ORA-600 [kksscl-inf-inl-loop]，具体如下：

```
Wed Apr 12 12:50:13 2006
Errors in file /oracle/app1/oracle/product/9.2/admin/bsssz/udump/bsssz_ora_5707.trc:
ORA-00600: 内部错误代码, 参数: [kksscl-inf-inl-loop], [2500], [259], [259], [520], [520], [], []
Wed Apr 12 12:50:46 2006
Errors in file /oracle/app1/oracle/product/9.2/admin/bsssz/udump/bsssz_ora_20951.trc:
ORA-00600: 内部错误代码, 参数: [kksscl-inf-inl-loop], [2500], [259], [259], [520], [520], [], []
Wed Apr 12 12:50:57 2006
Errors in file /oracle/app1/oracle/product/9.2/admin/bsssz/udump/bsssz_ora_21001.trc:
ORA-00600: 内部错误代码, 参数: [kksscl-inf-inl-loop], [2500], [259], [259], [520], [520], [], []
Wed Apr 12 12:55:20 2006
Errors in file /oracle/app1/oracle/product/9.2/admin/bsssz/udump/bsssz_ora_19245.trc:
ORA-00600: 内部错误代码, 参数: [kksscl-inf-inl-loop], [2500], [257], [258], [517], [517], [], []
Wed Apr 12 12:59:27 2006
Errors in file /oracle/app1/oracle/product/9.2/admin/bsssz/udump/bsssz_ora_16122.trc:
ORA-07445: 出现异常: 核心转储 [LE_sym_name()+100] [SIGQUIT] [unknown code]
[0x5C5E0000006A] [] []
ORA-01403: 未找到数据
Wed Apr 12 12:59:34 2006
Errors in file /oracle/app1/oracle/product/9.2/admin/bsssz/udump/bsssz_ora_20951.trc:
ORA-07445: 出现异常: 核心转储 [__doprnt_wide()+688] [SIGQUIT] [unknown code]
[0x5C5E0000006A] [] []
ORA-00600: 内部错误代码, 参数: [kksscl-inf-inl-loop], [2500], [259], [259], [520], [520], [], []
Wed Apr 12 13:00:02 2006
ORA-000060: Deadlock detected. More info in file /oracle/app1/oracle/product/9.2/
admin/bsssz/udump/bsssz_ora_16116.trc.
```

通过分析这个错误日志，初步感觉这个错误可能不是导致问题的原因，因为这个错误是由于 SQL 分析时等待子游标。等待子游标的父游标都有或多或少的子游标，多的有 200～300 个子游标，少的有 5～10 个子游标。关于这个问题我和李工进行过多次讨论，我又查阅了大量资料，初步的结论是这个故障可能是由于系统挂住引起的而不是导致系统挂住的原因。

第二次发生故障时临近下班，由于业务量不大，所以我和李工决定稍微等一下，暂时不杀掉

会话。等了大概五六分钟，系统挂住的状态还是没有改变，我和李工沟通后决定做一个 kill-3 操作。该操作给我们提供了一个有趣的信息，也最终为找到问题原因提供了最有利的数据。有趣的是 kill-3 不能停止该死锁进程的操作，不过能在日志中看到该 kill 操作被该进程捕获，说明该进程并没有死或者僵死，还在做转储操作。

通过该进程的跟踪开始时间和捕获到 kill-3 操作的时间可以看出，该进程写 Trace 文件已经超过 10 分钟了，还没有完成。是什么原因导致一个几兆字节的 Trace 文件 10 多分钟还没有完成呢？很明显，这个时候出现了内部死锁，而什么原因会产生内部死锁呢？通过跟踪，我们看到，Trace 文件里的会话持有了 parent enqueue hash chains，这个闩锁在写 Trace 文件时一直没有释放，而 parent enqueue hash chains 正是可能导致系统挂住的一个闩锁，因此可以判断系统挂住的主要原因是由于 parenet enqueue hash chains 闩锁被持有，而持有该闩锁的会话又由于写 Trace 文件十分缓慢而很久都没有释放该闩锁。

似乎问题的原因找到了，但是为什么写 ORA-60 Trace 时会持有 parent enqueue hash chains 闩锁呢？ORA-60 Trace 为什么会这么久都没有写完呢？只有搞明白了这一点，才能够真正找到解决这个问题的方法。

看样子有必要深入分析一下 Oracle 死锁处理的算法了，只有通过算法才能搞明白这几个问题。要了解算法，看样子又要用我蹩脚的英语去和远在美国的 John 交流了。虽然用英语表达对于我来说很痛苦，但是和 John 的交流还是很愉快的，每次都会带给我不少的收获。现在是晚上 8 点多钟，John 可能正在上班的路上，还是等明天白天再和他联络吧。不过如果现在发一个邮件给他，也许明天我醒来时，已经有一份答案在我的邮箱里了。

回到酒店时已经是晚上 10 点多了，通过 Skype 和老熊、老于通了通话，他们那边一切都很顺利。和我想象的一样，老熊和老于有一种一见如故的感觉，当然晚上也一起喝了点白酒。下午到达现场后，老熊已经开始着手分析那个我认为最为棘手的 SQL 了。在机场的时候我建议老熊先看看别的 SQL，刚上来就啃硬骨头，蹦了牙就不好玩了。我做项目有个习惯，一般来说先从容易的入手，一方面可以很快获得一些成果，给客户一定的信心，一方面也是偷懒，希望不用动最难的部分，就能够达成优化的目的。老熊的态度很明确，这个系统如果不解决几个主要的 SQL，优化效果是要打折扣的，早晚要碰，还不如现在趁着刚刚加入体力好、心里负担轻时先搞，否则过段时间，随着意志消磨，可能连碰这些 SQL 的心思都没有了。

既然有老熊这种好同志在前方这么玩命，我也可以安心地完成江苏的工作，不用急着赶回沈阳了。今天看样子能睡个好觉了。

今日点评

今天涉及的问题属于比较深入的问题，对于初级 DBA 来说就当看个故事吧。要想解决类似今天的问题，需要深入了解 Oracle 内部的一些算法，这也是很多 DBA 梦寐以求的事情。不过这些算法涉及 Oracle 公司的核心机密，因此也很难获取到。不过从 Metalink 上的一些 Oracle 内部

文档（标识为 internal only 的文档），我们可以从只言片语中获得一些信息，将这些算法还原出来。在写本书的同时，老白也在粉丝网上发了一些帖子，介绍过一些算法，大家有兴趣的话可以去查阅。

了解这些内容的方法无外乎几个途径。第一是更为深入地阅读 *Oracle Concepts*，从这本书里其实可以学到很多 Oracle 的基本思想和概念，勤奋的人也可以从中猜测出一些算法的思路。第二是通过 DSI 培训教材来了解 Oracle 内部的原理，DSI 是 Oracle 内部培训教材，不过现在大量的 DSI 教材已经流传出来了，特别是基于 Oracle 9i 的 DSI 4*xx* 系列，使用 Google 去搜索一下“DSI Oracle”，应该能找到不少资料。第三条途径是通过一些 Oracle 内部文档（标识为 internal only 的文档），这部分文档比 DSI 更难找到，不过有心人还是会有所发现的。其实在老白的 Oracle 粉丝网上就能找到不少这种文档。

国外几个网站上也有不少可以了解Oracle内部原理的资料，比如最为著名的 http://www.ixora.com.au 网站，这个网站上的资料虽然旧一些，不过确实都是很经典的。另外 asktom.oracle.com 也是一个很好的网站，有些不明白的地方，和 Tom 大师[1]聊聊，也许会有收获。

优化小技巧 Oracle 的死锁

死锁处理是 Oracle 锁处理机制中的一个亮点。一般来说死锁是很难避免的，除非在应用访问数据时做了十分严格的定义，对事务访问的数据源进行分级，按照级别进行访问。由于在应用开发时避免死锁十分困难，所以绝大多数应用都是依靠 Oracle 的死锁处理机制来处理死锁。Oracle 会自动进行死锁检测，一旦发现死锁后，自动解开。应用程序可能会碰到一个 ORA-60 的错误信息，那就是 Oracle 的死锁检测机制起作用了。

在 RAC 环境下，需要做全局的死锁检测，由 LMD 进程负责死锁检测。LMD 进程检测死锁的周期由参数`_lm_dd_interval` 确定，这个参数的默认值是 60s（大多数平台下）。

分析 ORA-60 Trace 文件时，主要是检查 WFG（Wait-For-Graph，等待图）。针对全局死锁，局部 WFG 的格式如下：

```
Global Wait-For-Graph(WFG) at ddTS[0.1] :
BLOCKED c000000007a2fcd8 5 [0xf002a][0x4bf],[TX] [65553,2]  0
BLOCKER c00000004ba2b090 5 [0xf002a][0x4bf],[TX] [131089,2] 1
BLOCKED c00000004ba2b4b0 5 [0x70022][0x589],[TX] [131089,2] 1
BLOCKER c000000007930470 5 [0x70022][0x589],[TX] [65553,2]  0
```

其中第一列列出了阻塞者（`BLOCKER`）和被阻塞者（`BLOCKED`），第二列是锁的地址指针，第三列是锁的模式，第四列是资源的 ID 及名字，第五列的含义是`<pid|did|txn_id>`，可能是 PID，也可能是进程事务号，最后一列是 RAC 节点的 ID，0 代表一号节点，1 代表二号节点。

锁模式的取值及含义说明如下。

- 0：NULL。

① 指 Tom Kyte。

- 1：并发读（SS)。
- 2：并发写(SX)。
- 3：保护读(S)。
- 4：保护写(SSX)。
- 5：排他访问(X)。

对于普通的 WFG，其格式和局部 WFG 略有不同，不过其定义比较清晰。下面是一个例子：

```
Deadlock graph:
                       ---------Blocker(s)--------  ---------Waiter(s)---------
Resource Name          process session holds waits  process session holds waits
TX-00020012-0000025e     12      11     X               11      10           X
TX-00050013-0000003b     11      10     X               12      11           X
session 11: DID 0001-000C-00000001      session 10: DID 0001-000B-00000001
session 10: DID 0001-000B-00000001      session 11: DID 0001-000C-00000001
Rows waited on:
Session 10: obj - ROWID = 00000BF6 - AAAAv2AAEAAAAqKAAB
Session 11: obj - ROWID = 00000BF6 - AAAAv2AAEAAAAqKAAA
```

优化小技巧　几个常用的与锁相关的脚本

本节将介绍一些 Metalink 上十分有用的锁使用情况，供大家参考。

1. 会话锁使用情况

使用会话锁时，其查询语句如下：

```
set linesize 200 feedback off heading on
column sid format 999
column res heading 'Resource Type' format a20
column id1 format 9999999
column id2 format 9999999
column lmode heading 'Lock Held' format a14
column request heading 'Lock Req.' format a14
column serial# format 99999
column username  format a10
column terminal heading Term format a6
column tab format a10
column owner format a8
SELECT  l.sid,s.serial#,s.username,s.terminal,
        decode(l.type,'RW','RW - Row Wait Enqueue',
                     'TM','TM - DML Enqueue',
                     'TX','TX - Trans Enqueue',
                     'UL','UL - User',l.type||'System') res,
        substr(t.name,1,10) tab,u.name owner,
        l.id1,l.id2,
        decode(l.lmode,1,'No Lock',
               2,'Row Share',
               3,'Row Exclusive',
               4,'Share',
               5,'Shr Row Excl',
               6,'Exclusive',null) lmode,
```

```
        decode(l.request,1,'No Lock',
               2,'Row Share',
               3,'Row Excl',
               4,'Share',
               5,'Shr Row Excl',
               6,'Exclusive',null) request
FROM v$lock l, v$session s,
sys.user$ u,sys.obj$ t
WHERE l.sid = s.sid
AND s.type != 'BACKGROUND'
AND t.obj# = l.id1
AND u.user# = t.owner#
/
set feedback on
```

2. 详细分解锁的情况

详细分解锁时，其查询语句如下：

```
set lines 200
set pagesize 66
spool locks.lis
break on Kill on sid on  username on terminal
column Kill heading 'Kill String' format a13
column res heading 'Resource Type' format 999
column id1 format 9999990
column id2 format 9999990
column locking heading 'Lock Held/Lock Requested' format a40
column lmode heading 'Lock Held' format a20
column request heading 'Lock Requested' format a20
column serial# format 99999
column username  format a10  heading "Username"
column terminal heading Term format a6
column tab format a30 heading "Table Name"
column owner format a9
column LAddr heading "ID1 - ID2" format a18
column Lockt heading "Lock Type" format a40
column command format a25
column sid format 990
rem     L.SID||','||S.SERIAL# Kill,

SELECT
nvl(S.USERNAME,'Internal') username,
        L.SID,
        nvl(S.TERMINAL,'None') terminal,
        decode(command,
0,'None',decode(l.id2,0,U1.NAME||'.'||substr(T1.NAME,1,20),'None')) tab,
decode(command,
0,'BACKGROUND',
1,'Create Table',
2,'INSERT',
3,'SELECT',
4,'CREATE CLUSTER',
5,'ALTER CLUSTER',
6,'UPDATE',
```

```
7,'DELETE',
8,'DROP',
9,'CREATE INDEX',
10,'DROP INDEX',
11,'ALTER INDEX',
12,'DROP TABLE',
13,'---',
14,'---',
15,'ALTER TABLE',
16,'---',
17,'GRANT',
18,'REVOKE',
19,'CREATE SYNONYM',
20,'DROP SYNONYM',
21,'CREATE VIEW',
22,'DROP VIEW',
23,'---',
24,'---',
25,'---',
26,'LOCK TABLE',
27,'NO OPERATION',
28,'RENAME',
29,'COMMENT',
30,'AUDIT',
31,'NOAUDIT',
32,'CREATE EXTERNAL DATABASE',
33,'DROP EXTERNAL DATABASE',
34,'CREATE DATABASE',
35,'ALTER DATABASE',
36,'CREATE ROLLBACK SEGMENT',
37,'ALTER ROLLBACK SEGMENT',
38,'DROP ROLLBACK SEGMENT',
39,'CREATE TABLESPACE',
40,'ALTER TABLESPACE',
41,'DROP TABLESPACE',
42,'ALTER SESSION',
43,'ALTER USER',
44,'COMMIT',
45,'ROLLBACK',
46,'SAVEPOINT',
47,'PL/SQL EXECUTE',
48,'SET TRANSACTION',
49,'ALTER SYSTEM SWITCH LOG',
50,'EXPLAIN',
51,'CREATE USER',
52,'CREATE ROLE',
53,'DROP USER',
54,'DROP ROLE',
55,'SET ROLE',
56,'CREATE SCHEMA',
57,'CREATE CONTROL FILE',
58,'ALTER TRACING',
59,'CREATE TRIGGER',
60,'ALTER TRIGGER',
```

```
61,'DROP TRIGGER',
62,'ANALYZE TABLE',
63,'ANALYZE INDEX',
64,'ANALYZE CLUSTER',
65,'CREATE PROFILE',
66,'DROP PROFILE',
67,'ALTER PROFILE',
68,'DROP PROCEDURE',
70,'ALTER RESOURCE COST',
71,'CREATE SNAPSHOT LOG',
72,'ALTER SNAPSHOT LOG',
73,'DROP SNAPSHOT LOG',
74,'CREATE SNAPSHOT',
75,'ALTER SNAPSHOT',
76,'DROP SNAPSHOT',
84,'-',
85,'TRUNCATE TABLE',
86,'TRUNCATE CLUSTER',
87,'-',
88,'ALTER VIEW',
89,'-',
90,'-',
91,'CREATE FUNCTION',
92,'ALTER FUNCTION',
93,'DROP FUNCTION',
94,'CREATE PACKAGE',
95,'ALTER PACKAGE',
96,'DROP PACKAGE',
97,'CREATE PACKAGE BODY',
98,'ALTER PACKAGE BODY',
99,'DROP PACKAGE BODY',
command||' - ???') COMMAND,
        decode(L.LMODE,1,'No Lock',
                2,'Row Share',
                3,'Row Exclusive',
                4,'Share',
                5,'Share Row Exclusive',
                6,'Exclusive','NONE') lmode,
        decode(L.REQUEST,1,'No Lock',
                2,'Row Share',
                3,'Row Exclusive',
                4,'Share',
                5,'Share Row Exclusive',
                6,'Exclusive','NONE') request,
l.id1||'-'||l.id2 Laddr,
l.type||' - '||
decode(l.type,
'BL','Buffer hash table instance lock',
'CF','Cross-instance function invocation instance lock',
'CI','Control file schema global enqueue lock',
'CS','Control file schema global enqueue lock',
'DF','Data file instance lock',
'DM','Mount/startup db primary/secondary instance lock',
'DR','Distributed recovery process lock',
```

```
'DX','Distributed transaction entry lock',
'FI','SGA open-file information lock',
'FS','File set lock',
'IR','Instance recovery serialization global enqueue lock',
'IV','Library cache invalidation instance lock',
'MB','Master buffer hash table instance lock',
'MM','Mount definition gloabal enqueue lock',
'MR','Media recovery lock',
'RE','USE_ROW_ENQUEUE enforcement lock',
'RT','Redo thread global enqueue lock',
'RW','Row wait enqueue lock',
'SC','System commit number instance lock',
'SH','System commit number high water mark enqueue lock',
'SN','Sequence number instance lock',
'SQ','Sequence number enqueue lock',
'ST','Space transaction enqueue lock',
'SV','Sequence number value lock',
'TA','Generic enqueue lock',
'TD','DDL enqueue lock',
'TE','Extend-segment enqueue lock',
'TM','DML enqueue lock',
'TT','Temporary table enqueue lock',
'TX','Transaction enqueue lock',
'UL','User supplied lock',
'UN','User name lock',
'WL','Being-written redo log instance lock',
'WS','Write-atomic-log-switch global enqueue lock',
'TS',decode(l.id2,0,'Temporary segment enqueue lock (ID2=0)',
                'New block allocation enqueue lock (ID2=1)'),
'LA','Library cache lock instance lock (A=namespace)',
'LB','Library cache lock instance lock (B=namespace)',
'LC','Library cache lock instance lock (C=namespace)',
'LD','Library cache lock instance lock (D=namespace)',
'LE','Library cache lock instance lock (E=namespace)',
'LF','Library cache lock instance lock (F=namespace)',
'LG','Library cache lock instance lock (G=namespace)',
'LH','Library cache lock instance lock (H=namespace)',
'LI','Library cache lock instance lock (I=namespace)',
'LJ','Library cache lock instance lock (J=namespace)',
'LK','Library cache lock instance lock (K=namespace)',
'LL','Library cache lock instance lock (L=namespace)',
'LM','Library cache lock instance lock (M=namespace)',
'LN','Library cache lock instance lock (N=namespace)',
'LO','Library cache lock instance lock (O=namespace)',
'LP','Library cache lock instance lock (P=namespace)',
'LS','Log start/log switch enqueue lock',
'PA','Library cache pin instance lock (A=namespace)',
'PB','Library cache pin instance lock (B=namespace)',
'PC','Library cache pin instance lock (C=namespace)',
'PD','Library cache pin instance lock (D=namespace)',
'PE','Library cache pin instance lock (E=namespace)',
'PF','Library cache pin instance lock (F=namespace)',
'PG','Library cache pin instance lock (G=namespace)',
'PH','Library cache pin instance lock (H=namespace)',
```

```
'PI','Library cache pin instance lock (I=namespace)',
'PJ','Library cache pin instance lock (J=namespace)',
'PL','Library cache pin instance lock (K=namespace)',
'PK','Library cache pin instance lock (L=namespace)',
'PM','Library cache pin instance lock (M=namespace)',
'PN','Library cache pin instance lock (N=namespace)',
'PO','Library cache pin instance lock (O=namespace)',
'PP','Library cache pin instance lock (P=namespace)',
'PQ','Library cache pin instance lock (Q=namespace)',
'PR','Library cache pin instance lock (R=namespace)',
'PS','Library cache pin instance lock (S=namespace)',
'PT','Library cache pin instance lock (T=namespace)',
'PU','Library cache pin instance lock (U=namespace)',
'PV','Library cache pin instance lock (V=namespace)',
'PW','Library cache pin instance lock (W=namespace)',
'PX','Library cache pin instance lock (X=namespace)',
'PY','Library cache pin instance lock (Y=namespace)',
'PZ','Library cache pin instance lock (Z=namespace)',
'QA','Row cache instance lock (A=cache)',
'QB','Row cache instance lock (B=cache)',
'QC','Row cache instance lock (C=cache)',
'QD','Row cache instance lock (D=cache)',
'QE','Row cache instance lock (E=cache)',
'QF','Row cache instance lock (F=cache)',
'QG','Row cache instance lock (G=cache)',
'QH','Row cache instance lock (H=cache)',
'QI','Row cache instance lock (I=cache)',
'QJ','Row cache instance lock (J=cache)',
'QL','Row cache instance lock (K=cache)',
'QK','Row cache instance lock (L=cache)',
'QM','Row cache instance lock (M=cache)',
'QN','Row cache instance lock (N=cache)',
'QO','Row cache instance lock (O=cache)',
'QP','Row cache instance lock (P=cache)',
'QQ','Row cache instance lock (Q=cache)',
'QR','Row cache instance lock (R=cache)',
'QS','Row cache instance lock (S=cache)',
'QT','Row cache instance lock (T=cache)',
'QU','Row cache instance lock (U=cache)',
'QV','Row cache instance lock (V=cache)',
'QW','Row cache instance lock (W=cache)',
'QX','Row cache instance lock (X=cache)',
'QY','Row cache instance lock (Y=cache)',
'QZ','Row cache instance lock (Z=cache)',
'TO','DDL operations on temporary tbl',
'DL','Direct Loader Index Creation',
'IN','Instance Number',
'IS','Instance State',
'KK','Redo Log Kick',
'PF','Password File',
'PI','Parallel Slaves',
'PR','Process Startup',
'PS','Parallel Slave Synchronization',
'TS','Temporary Segment (also TableSpace)',
```

```
'RO','Multiple object resue',
'????') Lockt
FROM    V$LOCK L,
        V$SESSION S,
        SYS.USER$ U1,
        SYS.OBJ$ T1
WHERE   L.SID = S.SID
AND     T1.OBJ#  = decode(L.ID2,0,L.ID1,1)
AND     U1.USER# = T1.OWNER#
AND     S.TYPE != 'BACKGROUND'
ORDER BY 1,2,5
/
```

5月20日 凌晨的邮件通知短信

昨天睡觉时忘记关手机了。我们的外国同行一般都会使用两个手机号码，下班后就关闭工作号码，认真地享受生活了。在国内就完全不一样了，做服务的人，手机基本上是不能关机的。以前公司人少的时候，我和老熊两个人如果都要从深圳飞北京，就要分坐两个航班，等老熊到达目的地了，我再出发，这样无论如何客户都能找到我们中的一个。我们几个人安排了值班制度，轮到值班时，会把服务热线转到值班人员的手机上，这时这个人的手机就必须 24 小时开机。这个月不是我值班，按理说是可以关手机的，由于 10 点多才匆匆住下，睡觉前居然忘记了。其实我关机的时候很少，哪怕是不值班的时候，一般也是 24 小时开着的。只有在长途旅行后或者加夜班后才会关闭手机，以保证睡眠。

这几天没有好好休息，在飞机上又看了半天文件，所以晚上不到 11：00 我就睡了。半夜，正在睡梦中的我被一阵手机铃声惊醒，第一时间我就觉得是 John 回信了，急忙拿起放在床头的手机。一看，却是一个房地产广告："万科东方尊峪，超低价发售，9999 起。"我失望地放下手机，正准备躺下，手机铃声再次响起，可恶的房产广告，可恶的万科，我正准备掐掉铃声，突然发现这是我邮箱的邮件通知短信。

John 真是个好同志，自从离开 Oracle 公司后，John 成为一家银行的 DBA 经理，空闲的时间比在 Oracle 时多了很多。用他的话说，从一个救火队员变成了一个悠闲的海滨度假者，从 40 岁开始，他要开始享受生活了，带小孩周游列国和为报纸撰写社评将成为他的主要工作。我曾经为此嘲笑他说，周游列国我相信，写社评我绝对不信，顶多也就给 Oracle 技术通信写几个客户来稿。不过自从他离开 Oracle 公司后，由于闲暇时间比较多，John 回答我的问题也越来越及时。我已经有一个多月没和他联络过了，估计这哥们也早就心里痒痒了。对 John 来说，有 Oracle 方面的难题给他，是最愉快的事情。

放下手机，我马上打开正在待机的电脑，查看 John 的邮件。John 的回答很短，显然是在匆忙中写的，翻译成中文就是："死锁引起数据库挂住是一个老问题，开发人员已经起码处理了几年，但是还有很多 bug 没有解决。死锁检测的简要过程在文档 285270.1 中说得很清楚（见附件），你应该一看就明白了。其关键原因是写 Trace 的时候，需要进行进程状态转储，而转储完成之前，持有的 parent enqueue hash chains 闩锁是不释放的，因为死锁检测过程必须判定哪些资源的等待状态没有发生改变，才会对这些没有发生改变的资源进行检查，否则检查面就太广了。持有 parent enqueue hash chains 是这个问题的关键。"

在 Metalink 文档 285270.1 里对死锁检测做了一个简要的介绍，看过这份文档，再参考 John 的提示，仔细想想，在做死锁检测的时候，确实是必须持有 parent enqueue hash chains 闩锁的，因为死锁检测（DLD，DeadLock Detection）需要通过对每个资源做一个检查，才可能发现哪些资源出现了死锁。而如果对每个资源都进行一次死锁检查，那么开销就太大了，因为在一个大型系统中同时有几十万甚至上百万个资源都是很正常的。每个资源上面都有三条链：一条是持有者链，链接了所有持有这个资源的锁的情况；一条是等待转换状态链，链接了所有等待转换锁状态的请求；还有一条是等待锁定链，链接了所有等待锁的请求。如果不出现死锁，那么这三条链和资源的状态在很短的时间内肯定会发生变化，而死锁出现时，相关的资源肯定在很长时间里都不会发生变化。因此在检测死锁时，只需要对那些从上回检查后状态一直没有发生过变化的资源进行检查就可以发现死锁了。为了获取这些数据的一致性视图，在做死锁检测时就必须持有这个闩锁。死锁检测本身很快，所以一般情况下不会引起问题。不过，如果出现了死锁，要做死锁情况转储时就可能会出问题。系统状态转储或者进程状态转储被挂住的可能性也是存在的，这很可能是由于另外一个 bug 引起的，无论哪个 bug，关闭转储应该可以解决问题。由于在进程状态转储时，死锁检测会话会持有 parent enqueue hash chains 父闩锁，所以在这个时候，任何需要申请锁资源（包括内部锁）的操作都需要等待。由于游标分析需要申请 library cache lock，所以在这种情况下，游标分析无法进行。部分会话会报 ORA-600[kksscl-inf-inl-loop]故障。

看看手机，已经是早上 6 点多钟，澳大利亚已经是早上 8 点多了，Ben 可能已经起床。Ben 有早上上网阅读早新闻的习惯，希望能碰到他，和他聊聊这个问题。打开 Yahoo Pager，发现 Ben 已经在线了。Ben 听了问题的叙述后，也立即说这是 Oracle 的一个顽疾，虽然出了很多补丁，但是还有一些问题没有解决。Ben 在 Oracle 工作快 10 年了，作为澳洲 Oracle 的救火队员，他处理过 10 多个类似的案例，因此他十分肯定地说我的猜测是对的。同时，我从 Ben 那里拿到了一分关于 bug 2235386 的资料，这份资料比我在 Metalink 上看到的要详细得多。从中，我有了十分惊人的发现。

bug 2235386 里面详细介绍了 ORA-60 导致系统挂住的情况，里面的内容与我和 John 的想法基本一致，但是 patch 2235386 并没有解决这个问题。要解决这个问题，需要在做进程状态转储之前最好释放闩锁，而这是不可能的，因为这样会导致进程状态转储或者系统状态转储的信息不准确。因此在这个补丁里引入了 10027 和 10028 两个事件，通过设置这两个事件来开启或关闭进程状态转储或者系统状态转储。原来如此，patch 2235386 并不能解决问题，而仅仅提供了解决问题的方法，因此 Oracle 的工程师仅仅是打了补丁，而没有解决问题，必须打完补丁后，通过设置 10028 事件关闭系统状态转储，才能解决这个问题。在死锁出现时，进程状态转储对于客户来说是没有多大用途的，因此关闭后对客户分析问题影响不大。

忙完这些已经快 9 点了，虽然昨晚一夜没怎么睡，不过现在一点都没有觉得困。上午一上班，我就和李工通了电话，在电话里，把我的发现和他进行了沟通。李工也基本认同了我的分析，决定先找一个本地网来做做实验，如果能够解决问题，再全面推广。原本定于今天下午的关于 RAC 的交流，由于他们有事，押到明天进行。看样子今天可以休息一下了。

和李工通过电话，绷得很紧的神经松弛了下来，困意也就上来了，倒头睡了个把小时就被打扫房间的服务员吵醒了。愤怒地赶走了服务员后，发现再也无法入睡，于是只好起来，到丰富路的巷子里买了大饼卷油条吃了，然后随便到街上逛逛。南京不比苏州、上海，吃的方面没有那么精细，早餐一般还是传统的那几样东西，大饼卷油条是南京的传统早餐，有点类似于北方的煎饼果子，不过大饼不是摊出来的，而是油煎的，吃起来比较有嚼头。

今日点评

今天老白通过美国的 John 和澳大利亚的 Ben 来帮助查找一些资料。他们两个往往能够找到老白在国内很难拿到的资料。做 DBA 的，多交几个这样的朋友是十分必要的，其实可以通过在国外的论坛、OTN 网站、Metalink 网站等渠道，认识一些老外，通过技术交流，大家建立起联系。条件成熟了，就可以通过 MSN、Yahoo Pager 等工具直接进行交流了。老白还碰到过一个 DBA，这个 DBA 为了扩大交流的圈子，还花钱申请了一个 OUG（Oracle 用户组织，付费用户可以获得更多的服务）的账号。

另外一个需要注意的方面是，DBA 可能经常会碰到一些特殊的情况，客户需要你给出一个合理的解释。比如说在这个案例里，很多现象纠缠在一起，必须做出一个十分合理的解释。今天老白根据死锁检测的一些内部原理对伴随出现的 ORA-600 [kksscl-inf-inl-loop]做出了合理的解释。这些解释需要对理论相当了解。由于很多 Oracle 的内部原理和机制是不对外公开的，所以经常会碰到一些你很难解释的情况。碰到这种问题时，如果你无法做出合理的解释，那么千万不要勉强自己去解释什么，否则最终你可能在挑战客户的智商，很难自圆其说，让自己下不了台，有过几次这样的经历后，客户对你的信任程度就会大大下降。

优化小技巧　10027/10028 事件

由于死锁检测时会锁住 parent enqueue latch 并且会进行进程状态转储，所以对于大型系统，包含状态对象比较多的情况下，这个转储工作可能会很慢，会导致系统短暂地被挂住。如果同时有大量的死锁出现，会导致系统变得十分缓慢，甚至出现闩锁死锁，导致系统长时间被挂住。解决办法是设置 10027 事件。

10027 事件从 Oracle 9.2.0.3 和 Oracle 10g 开始就已经支持了。而对 Oracle 9.2.0.2 和 Oracle 8.1.7.4，可以通过 Patch 2235386 来支持。10028 事件的 Level 定义如下。

- `0x00`：保持原来的状态，转储进程状态。
- `0x01`：关闭进程状态转储。
- `0x02`：完成进程状态转储后再转储系统状态。
- `0x03`：只转储系统状态。

10027 事件既可以关闭进程状态转储，使死锁检查变得更为快捷，以避免系统被挂住，又可以在需要对死锁进行更为详细的分析时加强转储的级别。

优化小技巧　PL/SQL 优化工具 profiler

在 Oracle 8i 中，PL/SQL 的效率得到了较大的提高。不过 PL/SQL 的性能提高在不同的硬件环境下是不同的，因此需要在实际运行环境中对 PL/SQL 的性能进行分析。DBMS_PROFILER 软件包可以帮助用户测试 PL/SQL 的性能。

DBMS_PROFILER 软件包是 Oracle 8i 和 Oracle 9i 的标准配置软件包。在$ORACLE_HOME/rdbms/admin 目录下可以找到 profload.sql 和 proftab.sql 两个脚本。

首先在系统用户下安装 profload.sql 脚本，然后在要测试 PL/SQL 的用户下安装 proftab.sql 脚本。

安装好脚本后，就可以使用 profiler 工具进行 PL/SQL 的性能测试了。首先执行如下 PL/SQL 代码：

```
declare
    err number;
begin
   err:=DBMS_PROFILER.START_PROFILER (to_char(sysdate,'dd-Mon-YYYY hh:mi:ss'));
  --此处执行要测试的 PL/SQL 存储过程或者程序
   err:=DBMS_PROFILER.STOP_PROFILER ;
end;
/
```

执行完毕后，通过下面的查询找出刚才执行的 runid：

```
column RUN_COMMENT format a40 truncate;
SELECT runid, run_date, RUN_COMMENT FROM plsql_profiler_runs ORDER BY runid;
```

然后，用下列查询查出运行结果：

```
column unit_name format a15 truncate;
column occured format 999999 ;
column line# format 99999 ;
column tot_time format 999.999999 ;
SELECT p.unit_name, p.occured, p.tot_time, p.line# line,
       substr(s.text, 1,75) text
FROM
       (SELECT u.unit_name, d.TOTAL_OCCUR occured,
               (d.TOTAL_TIME/1000000000) tot_time, d.line#
         FROM plsql_profiler_units u, plsql_profiler_data d
         WHERE d.RUNID=u.runid and d.UNIT_NUMBER = u.unit_number
           AND d.TOTAL_OCCUR >0
           AND  u.runid= &RUN_ID) p,
       user_source s
 WHERE p.unit_name = s.name(+) and  p.line# = s.line (+)
 ORDER BY p.unit_name, p.line#;
```

下面是一个输出的例子：

```
UNIT_NAME     OCCURED  TOT_TIME  LINE  TEXT
------------ -------- ----------- ----- ------------------------------------
NUMBER_TEST      1     .000006    13   dummy:='end of PLSQL';
...
```

注意分析 TOT_TIME 列，其中包含的是该语句执行的时间。通过分析执行时间，可以制订详细的优化计划。

profiler 是一个十分重要的工具，建议大家一定要掌握，因为这个工具可以提供十分精细的分析，对于在优化工作中进行测试和验证帮助很大。老白在后面的优化工作中也会用到这个工具。

5月22日 ODS 系统和 RAC

江苏的客户在准备上线的 ODS 系统中准备使用 RAC。客户以前在计费账务系统中使用过 OPS 系统，由于应用软件没有针对 OPS 进行优化，所以 OPS 的使用经历并不愉快，一直受全局缓存争用问题的困扰。客户在两个不同的实例上分别跑计费和账务两个应用，但是两个实例需要同时更新一些热表，因此这些热表出现了十分严重的 global buffer busy cr 等待，严重的时候两个系统会变得十分慢。在每个月的业务高峰期，由于全局 Cache 的争用，经常导致系统响应时间成倍地增长。在经过了痛苦的经历后，无奈之下只能拆分 OPS 系统，并通过物化视图实现两套系统的数据共享。因此客户对于再次使用集群技术，有点“一朝被蛇咬，十年怕井绳”的感觉。

随着 ODS 系统大集中需求的提出，今后单实例环境肯定无法满足全省的业务需要，使用集群方案势在必行。如何建设一个可持续扩展的应用平台，是下一步 ODS 系统发展的重点。为了防止计费账务系统的经历再次重演，这回客户和开发商在技术方面做了充分的准备。特别是在数据字典的设计方面，充分考虑了今后系统扩展的需要。而本次技术交流的主要目的，就是探讨在新版本的 ODS 系统中使用 RAC 的可行性以及针对 RAC 和应用软件应该做的优化工作。

RAC 环境中的优化是 RAC 系统成功的关键。在国外，RAC 应用十分成熟，在系统设计的初期，大多数系统就已经充分考虑了 RAC 对性能的影响，因此在软件架构设计时就已经充分予以考虑。刚上线的时候，可能只是一台很小的 Windows 或者 Linux 服务器，随着业务增长，通过不断加入新的 RAC 节点来实现系统的扩展。因此在国外，超过 10 个节点的 RAC 应用比比皆是，甚至我还见到过超过 20 个节点的 Linux RAC 集群。而事实上，国内 RAC 应用得并不很成功，由于应用软件未针对 RAC 进行相关的设计和优化，最终导致 RAC 系统出现严重性能问题的情况随处可见。国内的用户一般来说更喜欢淘汰旧设备、采购更为强劲的新设备来实现容量的扩充，这是因为软件开发商并没有针对 RAC 进行特殊的设计，在编程的时候也没有充分考虑 RAC 应用环境的特点，使用 RAC 后，1+1<1 的事情时有发生，因此 RAC 在国内大多数客户那边发挥的主要作用并不是负载均衡和可扩展性，而是类似于 HA 的高可用性。

为了使 ODS 系统能够具有高可扩展性，客户和应用开发厂商决定在系统的底层根据 RAC 环境进行一系列有针对性的设计和优化，因此，我针对数据库、应用软件等方面提出了一系列的优化要点。

除了在应用设计方面做一定的设计，尽量避免两个节点同时修改同一个数据外，对于 RAC 系统，数据库方面也有很多需要优化的方法。首先，针对数据库层面，在 RAC 环境下，DB Cache

的命中率对系统的性能影响远大于单实例环境，因此有效提高 DB Cache 的命中率，在 CPU 和内存资源充足的情况下合理设置 DB Cache 启用多缓冲池等，都是提高 DB Cache 命中率的有效方法。客户以前的系统使用的是 Oracle 8i 的 OPS，这方面的问题更为严重，而由于计费账务系统经常要对大表做扫描操作，所以当时 DB Cache 的命中率经常会低于 90%，因此全局 Cache 出现问题也就不足为奇了。

另外，尽量减少 buffer busy waits，buffer busy waits 会严重影响数据访问的性能，这在 RAC 环境下会更为严重。对于热块冲突较为严重的系统，在升级为 RAC 之前尽可能进行针对性的优化十分重要。在一般情况下，表分区是有效减少热块冲突的很有效的方法。对于像电信这样的以本地网为运营核心的系统，很少有跨越本地网访问的情况，系统中大多数的数据访问以本地网为维度，因此按照本地网进行数据分区是行之有效的。在 RAC 部署的时候，一般来说不建议采用完全的负载均衡模式，而是通过本地网来进行负载分配。全省 13 个本地网，可以按照一定规则分布在 RAC 的不同实例上，这样的负载分布配合按照本地网的表分区，可以最大限度地减少全局的热块冲突。解决热块冲突的方法有很多，对于表上面的热块冲突，可以通过表分区来减少热块冲突，也可以将热表存放在块大小较小的表空间上，以减少热块冲突的机会。对于不是很大的表，可以通过使用 hash 簇表的方式来替代传统的表（传统的表又称为堆表，Heap Table），hash 簇表按照某个键值 hash 方式存放，可以将表打散，从而减少热块冲突。如果热块冲突是针对索引的，那么通过分区表的本地索引、反转键索引，或者将索引存放在较小块大小的表空间中等手段，可以减少索引上的热块冲突。

锁等待在单机环境下也许很正常，但是在 RAC 环境下，严重的锁等待可能会导致严重的性能问题，因此尽量减少锁的使用，尽量减少不必要的索引。尽量减少不必要的 SQL 分析，通过各种手段提高 library cache 和 row cache 的命中率，也可以减少锁等待。

现在的应用大量使用序列号（sequence）,在 RAC 环境下，需要合理使用序列号，加大序列号的缓冲，尽量使用 NO ORDER 方式来有效减少序列号带来的争用。RAC 环境下普通的序列号缓冲是不能满足要求的，将缓冲设置为几百兆甚至几千兆可以大幅度减少由于序列号争用引起的性能问题。如果主键是通过序列号产生的，增加序列号缓冲的大小，还有助于减少主键索引引起的热块冲突。

如果某个和顺序无关的主键是由序列号形成的，建议在主键中拼入实例号以防止索引中热块的产生。这需要对应用进行小幅修改，在生成主键时，需要获取实例号，并且将实例号和从序列号产生的序列合并在一起，形成主键，这样的主键会让不同实例产生的记录在索引里最大限度地打散，最大限度减少由于索引带来的热块冲突。同时，索引应尽量采用本地分区索引，减少实例间的索引块争用。为了实现上述目标，使用 GUID 替代序列号也是一种方法。

对于存在大量并发插入操作的表，尽量使用 ASSM（自动段空间管理）。如果没有使用 ASSM，那么应该使用多个 Freelist Groups，并且适当增加 freelists 参数。

只读表空间（Read Only Tablespace）对于绝大多数用户来说都只限于理论的学习，很少被用于实际生产系统。在 RAC 环境中，只读表空间的使用可以大大提高 RAC 系统的性能。对于只读数据或者周期性修改的数据，可以考虑放入只读表空间中以减少 global cache cr request 的等待。

我曾经处理过一个案例，Global Cache 存在较为严重的竞争，后来我采取了一系列优化后将问题解决了，其中就将最常用的一些参照表存放的一个表空间设置为只读，只有需要修改这些参照表时才临时设置成读写。

一般情况下，如果我们对 RAC 系统从应用体系、数据与应用的分布、数据库相关参数等方面进行了相关优化，2 节点 RAC 达到 1.5～1.8 倍单机处理能力还是难实现的。

就这几个问题，我和客户以及开发厂商的技术人员进行了一上午的讨论。讨论的问题相当深入，他们提出的很多问题，我都必须认真思考一下，才能给出合理的解释。午饭时间到了，大家还是意犹未尽，由于下午他们要向领导汇报，所以我们只好结束了讨论。最后，他们希望我推荐一些资料给他们，我给他们推荐了下面几个 Metalink 的文档：

- Note:226561.1 9iRAC Tuning Best Practices
- Note:226569.1 9iRAC Most Common Performance Problem Areas
- Note:226573.1 9iRAC Workload Characterization

我用 U 盘将这几个文档复制给他们了。中午和他们一起在食堂吃了顿工作餐，吃饭时他们对我正在沈阳做的优化项目十分感兴趣，也问了很多相关的问题。听了我的介绍，大家都觉得沈阳的事情简直就是一件不可能完成的任务。

优化小技巧 以开放的心态做服务

今天江苏的客户向老白要一些资料，老白用 U 盘拷给了客户。在一些朋友看来，老白拷给客户的资料都是很重要的武林秘笈，不应该这么随便交给客户。DBA 这个圈子比较保守，一般不愿意把自己的独门绝技传人。而老白做服务这么多年来，一直用一种开放的心态，对客户的要求有求必应。记得有一次给客户做一个巡检，结束后老白把文档拷在 U 盘上交给客户，客户犹豫了半天，问道：“你的那些脚本方便不方便留给我？”老白笑笑说：“在 U 盘里已经有这些脚本了，我们做完巡检，会把巡检过程全部记录下来，并且会把使用过的脚本全部提交给客户。”

老白服务的客户很多，有些客户都已经和老白打了五六年交道了，关系一直比较融洽，主要的原因就是老白从来不保守，客户需要什么资料就会毫无保留地提供。客户咨询什么问题，老白一般来说除了当面口头回答外，事后还会以很详细的报告提交。

用一个形象的说法就是，如果目前客户处于 *A* 点，我们做服务的处于 *B* 点，*B* 点高于 *A* 点，而我们做服务的一个很重要的任务是使客户的技术从 *A* 点向 *B* 点迈进，同时我们自己也在从 *B* 点向 *C* 点前进。当客户达到 *B* 点时，我们已经达到了 *C* 点，这时我们就可以帮助客户从 *B* 点向 *C* 点迈进，而我们也向更高的目标前进。如果是这样一种情况，我们就不会害怕客户的提高危及我们自己的饭碗。对于客户而言，我们永远是有价值的，可以帮助他们向更高的目标迈进。

如果做服务的没有开放的心态，把自己的那些招数保护得死死的，那早晚是会做死的。保守就很难进步，时间长了，你的价值就会越来越低。所以我觉得以开放的心态来做服务，让客户在你的帮助下提高，促使自己也不断进步才是根本之道。

优化小技巧　全局热块冲突

全局热块冲突是 RAC 平台经常出现的一种等待事件，这种等待如果比较严重的话，会对系统的性能产生十分重大的影响，甚至导致数据库被短时挂住。

全局热块冲突和普通的热块冲突类似，普通的热块冲突是在一个单节点上多个会话访问相同的数据块导致的等待事件。如果大量会话访问某个或某些特定的数据块，那么这些数据块被称为热块。相比单实例环境，全局热块冲突的危害更大，因为全局的热块需要在实例间进行传递。严重的全局热块冲突会导致系统性能急剧下降。

最典型的案例就是，如果某个应用要对某张表进行大批量的数据插入，而且插入的进程都跑在一个节点上，那么这个系统可能会出现一些 buffer busy waits 等待事件，不过不会产生很严重的性能问题。如果这个应用负载均衡方式分布在不同的节点上，那么从 STATSPACK/AWR 报告中，我们可能看到下面的情况：

```
Top 5 Timed Events                                          Avg %Total
~~~~~~~~~~~~~~~~~~                                         wait   Call
Event                                 Waits    Time (s)   (ms)   Time Wait Class
------------------------------ ------------ ----------- ------ ------ ----------
buffer busy waits                    17,149      12,743    743   37.3 Concurrenc
gc buffer busy                       21,057      12,328    585   36.1    Cluster
enq: HW - contention                 19,571       7,155    366   20.9 Configurat
CPU time                                          1,253            3.7
gc current block 2-way              207,726         525      3    1.5    Cluste
          -------------------------------------------------------------
```

对于这种情况，需要在应用的底层进行设计。比如有一种很常用的方法，就是在某张表中设计一个 `INSTANCE_ID` 字段，在插入数据时，每个实例插入的数据中都带有 `INSTANCE_ID`，然后按照 `INSTANCE_ID` 对这张表进行分区，两个实例之间的 gc buffer busy 争用就会大幅度地减少。下面就是通过这种方式优化后的 AWR 报告。

Event	Waits	Time(s)	Avg Wait(ms)	% Total Call Time	Wait Class
CPU time		1,334		92.3	
log file sync	554,591	334	1	23.1	Commit
log file parallel write	536,004	106	0	7.3	System I/O
gc cr multi block request	449,571	85	0	5.9	Cluster
wait for scn ack	417,500	73	0	5.1	Other

从上面的报告可以看出，gc buffer busy 消失了。

优化小技巧　PL/SQL 中数据类型的性能

在 PL/SQL 编程中，要注意数据类型的使用。PL/SQL 支持多种兼容的数据类型，选择适合

的数据类型对于 PL/SQL 的性能有很大的影响。下面是一个简单的例子：

```
create or replace procedure  testplsql is
 anumber number ;
 ainteger integer;
 anumber_10 number(10);
 apls_integer pls_integer;
 abinary BINARY_integer;
 err number;
begin
  anumber:=0;
  loop
    anumber:=anumber+1;
    ainteger:=ainteger+1;
    anumber_10:=anumber_10+1;
    apls_integer:=apls_integer+1;
    abinary:=abinary+1;
    exit when anumber>1500 ;
  end loop;
end;
/
```

这个例子中测试了 number、integer、number(n)、pls_integer、BINARY_integer 这几种整数类型的运行效率。

我们使用了 DBMS_PROFILER 进行了测试，代码如下：

```
declare
err number;
begin
err:=DBMS_PROFILER.START_PROFILER (to_char(sysdate,'dd-Mon-YYYY hh:mi:ss'));
    testplsql;
    err:=DBMS_PROFILER.STOP_PROFILER ;
end;
/
```

查询 profiler 表，其结果如下：

模块	执行时间	行号	语句
TESTPLSQL	.000350	9	anumber:=0;
TESTPLSQL	.358233	11	anumber:=anumber+1;
TESTPLSQL	.358399	12	ainteger:=ainteger+1;
TESTPLSQL	.363219	13	anumber_10:=anumber_10+1;
TESTPLSQL	.133521	14	apls_integer:=apls_integer+1;
TESTPLSQL	.498786	15	abinary:=abinary+1;
TESTPLSQL	.511632	16	exit when anumber>1500 ;

从测试结果可以看出，pls_integer 最快，其次是 number 和 integer 类型，最慢的是 BINARY_integer。产生这种差异的原因何在呢？pls_integer 是 PL/SQL 2.2 版本的标准整型类型，只要是整型变量，取值范围在－2 147 483 647 和+2 147 483 647 之间的整数都可以用 pls_integer 来表示。pls_integer 是直接和 C 语言的类型对应的，在执行过程中没有类型转换的开销，因此最快。

使用和 C 语言类型对应的类型来定义变量可以大大提高 PL/SQL 的效率。下面是 PL/SQL 类型和 C 语言类型的对应关系。

```
C 语言类型                 PL/SQL 参数              PL/SQL 返回值
char                      IN VARCHAR2              VARCHAR2
char *                    IN OUT VARCHAR2          VARCHAR2
double                    IN NUMBER                NUMBER
double *                  IN OUT NUMBER            NUMBER
float                     IN NUMBER                NUMBER
float *                   IN OUT NUMBER            NUMBER
int                       IN PLS_INTEGER           PLS_INTEGER
int *                     IN OUT PLS_INTEGER       PLS_INTEGER
long int                  IN PLS_INTEGER           PLS_INTEGER
long int *                IN OUT PLS_INTEGER       PLS_INTEGER
short                     IN PLS_INTEGER           PLS_INTEGER
short *                   IN OUT PLS_INTEGER       PLS_INTEGER
void *                    IN OUT                   ORA_FFI.POINTERTYPE
```

5月23日 实时 ODS

今天是本次江苏之行的最后一天，主要内容是讨论 ODS 系统中的数据采集技术。ODS 系统是企业信息化建设中的关键系统，是目前国内电信运营商为了提高服务水平、掌握营销全局的重要辅助系统。ODS 系统具有以下几个特点：

- 面向主题的；
- 集成的；
- 易变的；
- 明细的；
- 反映当前数据。

ODS 是企业数据架构中最为复杂的一种形态，既要满足数据事务操作要求，又要满足数据分析要求，因此 ODS 系统的建设也是一项十分具有挑战性的工作。根据数据刷新实时性的不同，我们可以把 ODS 系统划分为以下几类。

- Ⅰ类 ODS：数据延迟为 1～2 秒，实时或近似实时。
- Ⅱ类 ODS：数据延迟为 2～4 小时。
- Ⅲ类 ODS：数据延迟为 12～24 小时。
- Ⅳ类 ODS：数据仓库中部分决策分析数据回流至 ODS 中。

数据延迟时间越短，ODS 建设难度越高，其中Ⅰ类 ODS 的建设难度最高，成本也是最高的。但是Ⅰ类 ODS 的实时性，可以为企业决策提供最为及时的分析数据，因此Ⅰ类 ODS 往往是企业经营者最为需要的。由于Ⅰ类 ODS 的实时性，它对技术的要求与其他类型的 ODS 也有所不同。一般来讲需要用到一些特殊的技术，对于企业级用户来说，如何高效地对海量数据进行变更捕捉、变更传输是 ODS 系统中的关键问题。对于技术的高要求导致实施难度相当大，因此在国内已经在运行的 ODS 系统中，Ⅰ类 ODS 几乎很难看到。

客户的 ODS 建设的起点很高，在建设之初就提出了 ODS 系统的数据采集以Ⅰ类和Ⅱ类为主的数据采集思路，因此 ODS 系统建设的难度较大，对于数据采集的要求较高。ODS 系统的数据采集不仅仅是从业务系统中将数据复制到 ODS 系统中，要求更多的是捕捉到数据变化的情况，根据数据的变化来生成 ODS 所需要的原始数据，提交给各种分析应用使用。传统的数据采集方式都很容易实现数据复制，但是对于数据变更的捕捉能力不足。

由于 ODS 系统是面向业务的，所以随着业务的发展，ODS 系统数据采集的需求会不断变化。

这种易变的特性需要 ODS 的数据采集系统采用良好的架构，具有很好的扩展性。

对于 Oracle 数据库来说，ODS 系统的数据采集一般可以使用以下方式。

- 离线文件方式：将业务系统变化捕捉后形成文件形式，传输到 ODS 系统，导入 ODS 系统。
- ETL 数据泵抽取方式：通过工具，从生产系统抽取数据，导入 ODS 系统。
- 高级复制：通过 Oracle 高级复制技术将生产系统的数据复制到 ODS 系统。
- Oracle Streams：通过 Oracle Streams 捕捉生产系统的变化，并将逻辑变化记录（LCR）传输给 ODS 系统，由接口程序处理 ODS，并形成 ODS 系统的最终数据。

离线文件的传输是通过编程的方式将生产数据打包成一定格式的文件，通过 FTP 等手段传输到 ODS 系统，然后使用 SQL*Loader 等工具将数据导入 ODS 系统。这种方式需要通过专门的程序来对文件传输和数据入库过程进行监控，确保数据复制的正确性。

ETL 数据泵抽取技术由于采用对生产库系统进行扫描获取相应数据的方式，所以对生产系统的影响较大。这种方式比较适合于抽取频率较低（比如每天一次、每月一次等）的非实时数据的抽取。

高级复制技术本身不是一种数据抽取技术而是一种数据复制技术，其主要任务是将数据从生产数据库中复制到 ODS 数据库或者接口数据库。ODS 系统需要对这些数据进行分析，才能捕捉到需要抽取的内容。

Oracle Streams 技术既支持数据复制，也支持变更捕捉，是一种十分适合 ODS 系统数据采集的技术。Streams 可以通过 LogMiner 技术分析生产系统的日志，将所关心的变更信息采集出来，通过高级队列传输到 ODS 系统。这样 ODS 系统收到的不仅仅是简单的表或者视图，而是某个 ODS 系统所关心的数据的变化增量。通过这个变化增量，ODS 系统可以更为灵活地生成决策支持所需要的数据，而不需要采用低效的数据比对方法来确定当前数据发生了什么变化。

我们向客户推荐了 Streams，不过由于客户主要的数据库都是 Oracle 9i，部分还是 Oracle 8i，目前直接使用 Streams 技术还存在一定的问题，因此决定暂时使用过渡性方案，就是使用物化视图日志产生数据库表的变化矢量，然后通过应用程序直接对物化视图日志进行处理，通过物化视图获取相关的变更信息，处理完毕后，手工删除物化视图日志中相关的数据。

这种处理有两个要点。一个是标准的物化视图日志只是一个根据时间戳的记录变更记录，比如对某条记录做了 2 次更新，那么在日志里存放的只是变化的过程，而每次如何变化是不记录的，这和 Streams CDC 是不同的。这样的处理对于高级复制是足够的，因为复制只需要保证目标和源一致，而 ODS 系统很多情况下是对某种变化进行统计分析，并依据变化的实际内容产生一系列数据，不是简单的数据复制。因此每个物化视图日志都要根据业务的特点来进行设计，确保分析需要的所有字段都在日志中存在。

第二点是，不能通过标准的刷新过程来刷新物化视图，ODS 系统并不一定需要存在一个物化视图，应用程序对物化视图日志进行处理后，根据应用的要求进行处理，比如删除已经处理完的记录。

实际上物化视图是通过物化视图日志来实现物化视图的刷新的。如果一个物化视图需要快速

刷新，那么就必须创建物化视图日志。一旦创建了物化视图日志，除了会产生一个物化视图日志对象外，还会创建一张和物化视图日志名字相同的表，这张表用于记录主表的变化信息。实际上物化视图日志里的数据是由一个系统触发器生成的，当主表的数据发生变化时，这会触发这个触发器，将变化的情况写入物化视图日志。物化视图日志可以被 ODS 系统使用，通过读取物化视图日志，了解生产系统的数据变化，但是物化视图日志中能够记录的信息比较少，只有主键、操作模式、时间戳等信息，而 ODS 需要的数据变化的信息较难获取。虽然我们可以在创建物化视图日志时将某些字段包含进去，但是这样会增加物化视图日志的大小。如果记录的字段数量太多，会影响生产系统的性能。

不过目前这种情况下，只能通过物化视图日志来获取数据的变化量了，这是没有办法的办法。

虽然对于使用 Streams 存在一定的疑虑，不过 Streams 优秀的特性还是深深打动了每个人。最后李工希望我们能够给他们做一个 Streams 数据采集的测试。测试的方法是在模拟两个生产库和一个 ODS 库。在两个生产库上分别通过 Streams 采集变化记录，并通过高级队列传播到 ODS 库上。ODS 库收到变化记录后，先将数据复制到一张表中，并通过变化记录中的数据修改一张统计表中的数据。

当天下午，在南京办事处，我和小邓、小张一起进行了测试。我们直接通过 Oracle 9.2.0.8 的库进行实验。Oracle 公司建议使用 Oracle 9.2.0.6 以上的版本，因为手头只有 Oracle 9.2.0.8 的升级包，所以我们直接使用 Oracle 9.2.0.8。

首先我们模拟了一个 200 万数据的变化，在生产库上产生 200 万新数据，测试 ODS 库上接收和处理数据的情况。这个实验到晚上 11 点多才结束，测试十分成功。ODS 库上的数据延时在一分钟以内，生产库一切正常，并没有因为 Streams 受到严重的影响。

我让小邓第二天继续实验，测试如果 ODS 库的 Apply Handler 出问题后，生产库是否会受到影响。如果这个实验能够通过，我们甚至可以建议客户跳过物化视图日志，直接使用 Streams。

今日点评

Streams 技术是一项十分好的技术，随着 Oracle Streams 的不断成熟，今后完全可能替代目前的一些复制技术。基于 Streams 的复制可能会替代传统的物化视图。

在 Oracle 9.2.0.8 上，Streams 已经比较成熟了，虽然国内用户使用得还不多，在国外的应用案例比比皆是。老白这些年也实施了不少 Streams 的案例，Oracle 9i 上 Streams 的主要问题是它使用共享池作为缓冲区，很可能由于 Streams 出现流控，积压数据而影响共享池。因此，在共享池争用较严重的系统中或者系统负载比较高的系统中，要慎重使用。对于 10.2 版本的数据库，Streams 技术更为成熟，Streams 池完全独立了出来，因此 Streams 导致生产库故障的机会大大下降。只要打好相关补丁，一般来说，Streams 的运行还是相对平稳的。相比 DSG 等第三方复制产品来说，Streams 也一点都不逊色，在经过充分优化的情况下，其性能也是十分出色的，在一个每天产生 200GB～300GB 重做日志的系统中，通过流复制复制绝大多数表，如果优化得当，其

延时可以控制在 10 分钟以内。

不过对于广域网环境下的数据复制来说，Streams 可能由于广域网的网络性能不稳定而导致一些性能问题。Streams 不一定需要很高的带宽，但是对网络带宽的稳定性要求较高。因此，在广域网中使用 Streams，尽可能选择传输较为稳定的光纤专线或者数字专线。

使用 Streams 的朋友一定要阅读一下 Metalink 的文档 437838.1（Recommended Patch for Streams）、290605.1（Oracle Streams STRMMON Monitoring Utility）、752871.1（Streams Complete Reference FAQ）、746247.1（Troubleshoot and Resolve Streams Capture when it is Paused for flow control）、418755.1（10gR2 Streams Recommended Configuration）。

优化小技巧　ODS 系统中 Oracle 数据库设计管理要点

现在很多行业都在建设 ODS 系统，ODS 系统的主要特点是海量存储、数据变化快、分析统计型的查询比较多。因此，对于 ODS 系统的配置也有一些特殊的地方，本节的目的是探讨几个对于 ODS 系统今后发展十分重要的方面，这些方面应该在 ODS 设计时充分考虑。

由于 ODS 系统是易变的，所以 ODS 的数据采集框架一定要有比较好的设计，元数据的管理也要比较规范。ODS 系统的数据来源十分复杂，可能来自于不同的系统、不同的数据库，部分数据还可能来自于平面文件。因此，选择 ODS 数据采集产品和数据采集框架一定要十分慎重。Oracle 的 Streams 是一个不错的数据变化量采集框架，很多 DBA 都使用过 Streams，不过主要用途是数据复制。实际上，Streams 提供了十分优秀的 API，其基本原理是通过重做日志获得数据变化的记录，然后通过暂存转发机制，通过 AQ 传输到目的地，然后通过 Apply Handler 将变化量取出，进行处理后提交到目的数据库中。在实际使用时，我们可以使用这个框架通过 API 来处理各种情况。如果我们要从一个非 Oracle 数据库中获取变化量，然后提交到 Oracle 数据库中，比如源数据库是 SQL Server，那么可以在 SQL Server 服务器上获取到数据库的变化量，然后通过 Streams 的 API 将这些变化量放入 AQ 中，传输到目的数据库上，用目的数据库的 Apply Handler 取出变化量，进行处理后，写入目标数据库中。实际上，ODS 数据库需要的很多数据不是直接从原始数据库中复制数据，而是要通过原始数据库中的数据变化量来更新 ODS 库中的某些数据。这种情况下，捕捉变化量是十分关键的。通过 Streams 获取变化量，经过处理后再修改 ODS 数据的方式十分有效。

ODS 数据库中还有大量的数据是直接从生产库中复制过来的。复制这些数据的方法有很多，最常用的是物化视图。物化视图是一种十分高效和易于管理的数据复制技术，目前已被广泛应用。它在 ODS 系统中也是最常用的数据复制技术，任何一套 ODS 系统中都少不了它。ODS 系统采用的技术适合用就好，最忌讳的是片面强调框架和新技术，因为 ODS 系统是在动态中维护和管理的，所采用技术的维护成本是要充分考虑。

在设计 ODS 数据库时，一般建议使用 ASSM 技术。不过在 ODS 数据库中，也应该设计部分 MSSM 的表空间，因为 ASSM 表空间存在大量的 bug，可能导致 ORA-600 [kcbgtcr_xx]和 ORA-600 [kcbnew_xx]错误。对于那些经常要截断（truncate）后重新载入数据的表，一旦出现上述的 ORA-600

错误，那么可以将这些表转移到 MSSM 的表空间上。

在表空间的设计上，一般建议使用统一大小，这样可以避免今后出现表空间碎片的问题。在规划之初往往感觉 ODS 系统容量十分充足，不过运行一两年后，就会发现扩容赶不上数据的增长，这是 ODS 系统易变的特性决定的。随着 ODS 应用的深入，业务需求不断地增加，容量不足的问题很容易出现，如果再有大量的空间被浪费在表空间碎片中，那么就更不值了。使用统一扩展大小，会对应用部署设计带来一定的挑战。需要设计几种统一大小的表空间，每种表空间有几个，各有多大容量？这些都要预先设计好，并且要在数据库维护工作开展时不断调整。比如数据里是否存在 LOB 字段，这些 LOB 字段都是多大的，是否大多数不能 in-line 存储？这些对于数据字典的设计有很大的影响。如果 LOB 字段都很大，甚至有几千个字节，那么这种情况下将其和表放在相同的表空间肯定是不合适的。LOB 字段需要存放在比统一大小更大的表空间里，甚至需要存放在块更大的表空间里。

在 ODS 系统中使用不同的数据块大小是很正常的事情，这在 OLTP 系统中却很少使用。实际上使用不同大小的数据块无论对于 OLTP 系统还是对于 OLAP 系统，都是十分好的，不过由于目前国内软件开发往往注重功能性的实现，而较为忽视性能和维护方面的问题，所以在应用中很少使用多种数据块大小。在 ODS 系统中，可能存在某些索引需要存放在较小块大小的表空间里，而某些经常做全表扫描的表适合存放在较大块大小的表空间里。这些都需要 ODS 的设计者充分考虑，否则等到出现性能问题时再来调整，代价就十分高了，因为 ODS 系统除了易变之外，一个特点就是海量存储。

优化小技巧　ODS 系统中的开发要点

ODS 系统的数据是海量存储，数据抽取、导入、分析都会面对大量的数据。如何处理海量数据是一个恒久的话题。提高海量数据处理速度是 DBA 经常面对的问题。其实开发人员也需要补充一些海量数据处理的常识。下述要点就是写给开发人员的。

- 尽可能使用 `INSERT /*+ APPEND */`。在 ODS 系统中存在大量的 `INSERT…SELECT…`操作，我看到很多 ODS 应用在这种情况下都没有使用`/*+ APPEND */`提示。使用 `APPEND` 提示可以大大提高批量插入的性能。
- 批量提交：对于大批量数据的插入，应该使用批量提交。对于每次应提交多少条记录，没有一个明确的数字，它和表的行长度、数据库配置、重做日志性能等都有关系，不过一般来说批量应该在几百条以上。具体多大的批量好，通过实际试验测试才能有比较准确的结果。比如某个插入，可能 3 000 条记录时性能最好，超过 3 000 条，性能提升就不大，超过 5 000 条性能反而下降。
- bulk 操作：数组操作从 Oracle 7 开始就在 OCI 里支持了。无论是 DML 还是 `SELECT` 在大批量数据操作时，使用 bulk 操作都可以大幅度提高性能。如果使用的是 C 语言，Pro*C 和 OCI 都支持 bulk 操作；如果使用的是 PL/SQL，也可以通过 `FORALL` 操作来实现 bulk 操作。

- 充分考虑 undo 的承受能力：ODS 系统中经常要生成一些临时表，或者往临时表里写入数据。经常的操作是 INSERT INTO…SELECT，这种操作往往会消耗大量的 undo，甚至导致 undo 表空间不足。对于超大量数据的操作，最好能够分段进行以避免 undo 问题。

优化小技巧　集合类型的效率

集合类型的性能一定好吗？下面的例子用于分析集合类型的效率。

```
CREATE OR REPLACE TYPE big_emp_ename_t AS TABLE OF varchar2(10);
/

CREATE OR REPLACE TYPE big_emp_ename_v AS VARRAY(200002) of varchar2(10);
/

CREATE OR REPLACE PROCEDURE coll_test IS
  TYPE big_emp_ename_i IS TABLE OF varchar2(10) INDEX BY BINARY_INTEGER;
  big_emp_cache_t big_emp_ename_t;
  big_emp_cache_v big_emp_ename_v;
  big_emp_cache_i big_emp_ename_i;

  CURSOR big_emp_curs IS SELECT rownum, ename FROM big_emp;
  cnt number;
  ename varchar2(10);
  p pls_integer;
BEGIN
  big_emp_cache_t:=big_emp_ename_t(' '); --to initilaize
  big_emp_cache_v:=big_emp_ename_v(' '); --to initilaize

  FOR be_r IN big_emp_curs LOOP
    big_emp_cache_t(be_r.rownum):=be_r.ename;
    big_emp_cache_t.EXTEND;
    big_emp_cache_v(be_r.rownum):=be_r.ename;
    big_emp_cache_v.EXTEND;
    big_emp_cache_i(be_r.rownum):=be_r.ename;
    ename:='DUMMY';
  END LOOP;

  SELECT COUNT(*) INTO cnt FROM big_emp WHERE ename='SCOTT';

  SELECT COUNT(*) cache_t INTO cnt
    FROM TABLE(CAST( big_emp_cache_t AS big_emp_ename_t)) be_c
   WHERE be_c.column_value='SCOTT';

  SELECT COUNT(*) cache_v INTO cnt
    FROM TABLE(CAST( big_emp_cache_v AS big_emp_ename_v)) be_c
   WHERE be_c.column_value='SCOTT';

  ename:= big_emp_cache_t(100000);
  FOR i in 1..1000 LOOP
    p:=100000+i;
    ename:= big_emp_cache_t(p);
    ename:= big_emp_cache_v(p);
```

```
    ename:= big_emp_cache_i(p);
  END LOOP;

  ename:='DUMMY';
END coll_test;
```

下面是 profiler 的结果。

```
UNIT_NAME OCCURED   TOT_TIME LINE TEXT
--------- ------- ---------- ---- ---------------------------------------------
COLL_TEST       1    .000341    7   CURSOR big_emp_curs IS
                                        SELECT rownum, ename FROM big_emp;
COLL_TEST    1   .000064  12  big_emp_cache_t:=big_emp_ename_t(' '); --to initilaize
COLL_TEST    1   .000037  13  big_emp_cache_v:=big_emp_ename_v(' '); --to initilaize
COLL_TEST  200002  24.293033   15   FOR be_r IN big_emp_curs LOOP
COLL_TEST  200000   3.317925   16     big_emp_cache_t(be_r.rownum):=be_r.ename;
COLL_TEST  200000   2.196512   17     big_emp_cache_t.EXTEND;
COLL_TEST  200000   1.957762   18     big_emp_cache_v(be_r.rownum):=be_r.ename;
COLL_TEST  200000   1.446018   19     big_emp_cache_v.EXTEND;
COLL_TEST  200000   2.022522   20     big_emp_cache_i(be_r.rownum):=be_r.ename;
COLL_TEST  200001    .502567   21     ename:='DUMMY';
COLL_TEST       1    .706513   24   SELECT COUNT(*) INTO cnt FROM big_emp
                                           WHERE ename='SCOTT';
COLL_TEST       1   1.287485   26   SELECT COUNT(*) cache_t INTO cnt
COLL_TEST       1   1.333679   30   SELECT COUNT(*) cache_v INTO cnt
COLL_TEST       1    .000026   34   ename:= big_emp_cache_t(100000);
COLL_TEST    1001    .001445   35   FOR i in 1..1000 LOOP
COLL_TEST    1000    .007430   36     p:=100000+i;
COLL_TEST    1000    .005262   37     ename:= big_emp_cache_t(p);
COLL_TEST    1000    .004718   38     ename:= big_emp_cache_v(p);
COLL_TEST    1000    .006192   39     ename:= big_emp_cache_i(p);
COLL_TEST       2    .113792   42   ename:='DUMMY';
```

big_emp 表有 200 000 行记录，被装载进一个内存数组（VARRAY）、一个 PL/SQL 表和一个索引表。第 16～20 行的数据表明装载时间比较长，装载进索引表的时间最短，其次是 VARRAY，最慢的是 PL/SQL 循环表，比索引表慢 3 倍。

从第 37～39 行可以看出访问各种集合类型的速度。这 3 种类型之间的差别不是很大，VARRAY 是最快的。

从查询速度来看，从实际表中查询的速度最快，它比从内存表中查询还要快。这个例子并不能说明内存表查询慢，只是可以说明在某些情况下，内存表不一定是最佳的选择。

5月24日 任务提前

今天终于完成了南京的事情，我把 Streams 后续实验的任务交给小邓后，就坐飞机返回沈阳了。今天运气真不错，在机场居然补到了一个头等舱的位置。我拿的是深圳航空的金卡，可以免费升舱。由于我坐的航线很多没有头等舱，就算偶尔有头等舱，也经常由于我到机场比较晚，已经无法升舱了。这还是我第一次坐头等舱，以前最多也只是坐过商务舱，头等舱的位置确实舒服。

在做一个大项目时，最好是能够不被打断，持续地做下去。我离开的这段时间，沈阳这边发生了一系列事情，所以我需要做一些功课才能够跟得上进度。吃过丰盛的午饭，我就马上打开电脑，查看老于和老熊发过来的资料。我不在沈阳的一个星期，老于他们的进展十分迅速，主要的优化思路已经基本形成。25 日开始就是月底高峰了，25 日到下个月 10 日这段时间是我们分析系统的关键时段，在这个时段里我们必须把主要的问题都一一发现，为下一步形成优化方案打下坚实的基础。

从报告上看，老熊最近一直在优化一个通过查询管理器生成出来的 SQL,这个 SQL 占缓冲区获取的比重超过了 20%，如果能够对这个 SQL 进行优化，就可以大大减轻账务系统的 CPU 消耗。从执行计划上来看，由于这个 SQL 是由一个查询生成器生成的，而开发人员为了省事，使用了大量比如 `where 1=1 and 1=0 and ...`这样的占位操作和开关操作，并且这个 SQL 由 7 张表组成，其中 5 张是超过百万的表，3 张甚至是超过千万的表。经过几天的分析，老熊找到了解决这个问题的方法，只要去掉占位操作和 1=0、1=1 这样的开关，优化器给出的执行计划是十分正确的，整个 SQL 的开销减少了 95%以上。老熊发现了这个解决方案后，十分兴奋，立即和老于一起找来了开发厂商的技术人员，一起讨论这个优化方案。不过在开发商那边，又遇到了很大的阻力，经过讨论后，开发商拒绝做出修改，理由是如果去掉占位操作，程序的逻辑就复杂多了，他们的开发人员没有能力完成这个改造。这是一个很令人郁闷的结果，也是我们做优化过程中经常遇到的结果。应用开发人员最不愿意做的就是修改自己的应用。

看到开发厂商的态度十分坚决，老于和老熊也没有办法。后来通过私下交流才了解到，这个应用十分复杂，性能问题也是大家早就有所认知的，不过由于最初开发这个模块的人早就离开公司了，所以这个模块一直没有人敢去做大的改动，哪怕是有新的需求，也只敢通过占位符和开关变量增加几个开关而已。我回到沈阳时，老熊正在为这件事情而感到十分郁闷，老于也是满脸无奈。晚上凑在一起喝酒，气氛也不是很好。如何协调和应用开发厂商的关系十分关键。按照原来的计划，25 日开始是信息收集的关键阶段，我们要完成所有基线数据的采集，并且根据这段时

间收集的信息，在下个月 10 日前完成优化方案的设计。不过由于客户那边面临的压力越来越大，客户希望我们能够在 6 月 6 日完成第一次优化操作，以便顺利度过 6 月 8 日开始的缴费高峰。老于开始的时候也强调提前优化实施对整个项目的影响较大，希望能够按照原来的计划实施，不过由于孙主任那边实在是挺不住了，直接给小齐打电话，要求我们能够帮他们个忙，提前做一些调整。甚至北方公司的李总也找到小齐要求我们配合辽宁公司尽快做一次调整。

既然小齐已经答应了客户，那么我们就必须认真对待这件事了，我们必须为此调整工作计划。这种节奏的变化是优化工作中的大忌，按部就班地实施优化是十分必要的。由于整体优化项目不同于普通的优化，这是一个系统工程，每个步骤都依赖于前一个步骤的成果，如果优化实施提前，那么前面的每个步骤都要提前。虽然总体的优化思路，我和老于已经基本形成，但是其中的很多细节需要等这个月底和下个月初，甚至是 6 月 8 日以后从系统分析中获取，很多猜想也要通过后面半个多月的数据分析来验证。如果 6 月 6 日实施了部分优化操作，那么 6 月 8 日后的有些东西可能会被掩盖，这里面如果存在一个严重的问题被我们忽略了，那么就给系统埋下了一个定时炸弹，这个炸弹一旦爆炸就是致命的。如果爆炸时正好是 7 月 8 日我们准备验收的时候，那么 7 月份验收计划就会成为泡影。

还有一点我十分顾虑，那就是目前集成商的 SQL 优化工作基本在原地踏步，如果 6 月 6 日优化实施后，系统性能有所改善，那么集成商优化 SQL 的力度也会减弱，这样就更不利于推动集成商修改 SQL 了。这个问题老于今天也和甲方提出了，甲方的态度十分明确，就是集成商的问题我们不用管，他们会给他们足够的压力，不会因为 6 月 6 日系统性能改善而影响 SQL 优化工作。看样子 6 月 6 日的优化实施势在必行了，过多的坚持没有任何意义，因为客户目前的处境我们都很清楚。他们承受的压力比我们多好多倍，如果不是实在支持不住了，他们也不会不顾一切地要求我们提前实施。

晚上我们一边喝酒一边研究 6 月 6 日的优化要做哪些调整，由于 6 月 7 日我还要到大连开一个会，所以 6 日的优化工作必须滴水不漏。而 6 日之前让应用开发厂商完成部分应用的修改工作已经是不可能的了。我们在 6 月 6 日实施的优化项目中，采用的都是一些比较保守的方案，也就是副作用比较小的操作，所以我们计划 6 日的优化重点分为以下几个方面。

- 大表分区，对于超过 10GB 的大表，如果可能就进行分区。
- 调整部分索引。
- 对主要常用索引进行重建（通过 `ALTER INDEX <...> REBUILD` 或者 `ALTER INDEX <...> REBUILD ONLINE`）。
- 调整操作系统 VM 相关参数。
- 调整部分 Oracle 参数。
- 全面分析表和索引。

对于表分区技术的使用，我们几个都有十分丰富的经验，不过以往做表分区，都是在开发人员的配合下完成的，这一次我们将脱离开发人员的配合独立完成，所以风险会加大很多，并且需要我们花更多的精力来分析应用才能够完成。因为目前分区主键的选择以及分区粒度的设计完全

要依靠我们自己的经验来设计，所以分区主键选择的合理性对系统性能改善影响极大。

添加索引的问题，在之前已经通过 SQL Analyzer 分析了大量的 SQL，形成了几十个索引优化的建议。不过根据我们前一阵子对索引进行的监控发现，还有 1 000 多个索引从来没有用过，是否要删除这些索引的问题，就没那么容易解决了。虽然在我们监控的这段时间里，这些索引没有用过，然而这并不能说明这些索引都是没有任何用处的。假设每个月只有少量的操作用到了某个索引，而这个索引又被我们删除了，如果更不巧的是使用这个功能的人正好是领导，那么麻烦就大了。以我的经验，在优化项目实施过程中，这种存在一定风险的操作是要尽量避免的。从另一个角度上看，实际上用户感觉比较慢的操作或者消耗系统资源比较大的操作，往往是一些复杂的查询，而不是 INSERT 操作，乱建索引会增加 INSERT、DELETE 和 UPDATE 操作的开销，影响这些操作的性能。而从目前系统的整体情况来看，用户对这些操作的性能并不敏感，而且这些操作占系统总开销的比例不足 5%，因此我们没有必要冒着出问题的风险，第一时间删除这些索引。主要是因为目前开发商不能全力配合，导致这个原本很容易的问题变得复杂起来。通过协商，我们决定暂时只增加索引，而不做删除操作。

对于 VM 参数的调整，通过我们前一段时间的分析，已经发现 maxperm%等参数设置有明显的不合理现象，减少 maxperm%的值，可以将大量的内存从文件缓冲中释放出来，从而解决系统负载高时产生大量换页的问题。虽然说 Unix 操作系统的文件缓冲可以在需要时释放，不过由于 Unix 系统的内存释放算法存在问题，这种听起来很不错的机制其实是存在一定问题的。最好的方法是让文件缓冲尽可能少地占用物理内存，因为对于数据库服务器来说，文件缓冲几乎没有什么用处。

调整完 VM 相关参数后，物理内存会出现较多的空闲，这使我们能够加大数据库缓冲池、共享池等缓冲池的大小。我们决定加大这些缓冲池，但这个操作存在较大风险，因为在业务高峰期间，CPU 的使用率已经相当高了，甚至严重的时候已经出现了 CPU 的瓶颈。我们做出调整这些缓冲池的决定基于综合考虑：一方面是通过索引的优化以及索引重建，提高索引的效率，减少全表扫描带来的系统开销；一方面是通过表分区减少表扫描以及索引扫描的开销；最后我们通过对表和索引进行重新分析，以提高基于成本的优化器（CBO）的效率。在这种情况下，我们才敢于在 CPU 存在明显瓶颈的系统上增加缓冲池的大小。另外，为了进一步减少 CPU 的开销，我们决定调整游标相关参数（包括 cursor_sharing、session_cached_cursors 等）来减少 SQL 分析带来的 CPU 开销。

今日点评

DBA 在做 SQL 优化时，经常会犯一个错误：找出一堆开销较大的 SQL，提交给开发商去修改。这是做优化的大忌。开发商需要的不是你提交一堆 SQL 给他们，而是需要将解决方案找出来交给他们。甚至在很多情况下，需要找出多个解决方案去供开发商选择。实际上，在一个优化项目中，开发商和 DBA 的目标应该是相同的，而往往由于两方代表的单位之间的利益冲突造成

了 DBA 和开发人员之间的激烈冲突。这种冲突在严重的时候会影响整个优化项目。

开发人员最怕的是 DBA 的优化计划给他们带来很大的工作量，因此在应用优化上会出现一定的抵触情绪。如果 DBA 交给开发人员的优化计划十分容易实现，那么这种抵触情绪会有所减弱。因此 DBA 在优化项目中，首先应该和开发人员建立起一种良好的合作关系，并在制订优化方案时尽可能地替开发人员考虑，选择一些比较容易实现的优化方案，在必要的时候需要提供多个可选方案供开发人员选择。如果能够做到这点，开发人员就没有任何借口来为自己开脱了，这对于整个优化项目是十分有帮助的。

优化小技巧 占位操作与开关操作

占位操作是开发人员常用的一种编程方法，比如 `WHERE` 条件是动态生成的，那么 `WHERE` 后面的第一个条件和更后面的条件是不同的，`WHERE` 后的第一个条件前面没有任何运算符，而后面的都带有运算符，因此如果 `WHERE` 变为 `WHERE 1=1`，那么后面的所有条件都有运算符，编程就简单很多。实际上占位操作是一种很不好的编程习惯，如果条件很复杂，又大量使用占位操作，那么会导致优化器无法获得正确的执行计划。

开关操作也是编程人员常用的一种编程手段，同样也是性能杀手。比如写好一个 SQL 模板，其中有一个开关（`1=:p1 and...`），如果我们希望后面的 `and` 起作用，那么 `p1` 就赋值为 1，如果不希望后面的 `and` 起作用，`p1` 就赋非 1 的值。

对较为复杂情况的应用来说，占位操作和开关操作可以提高 SQL 代码的重用，简化 SQL 的逻辑关系。但是占位操作可能带来严重的数据库性能问题。由于使用了大量的开关变量和占位符，所以优化器很容易产生错误的执行计划，增加 SQL 的成本。实际上，占位操作完全可以通过应用程序的逻辑来替代。在应用软件开发过程中，尽量不要为了满足复杂的逻辑关系而编写过于庞大的 SQL。通过程序的逻辑控制，将一个复杂的 SQL 分解为几个小型的 SQL 是一种十分好的编程习惯。由于开发团队往往过多地考虑功能上的实现以及代码的重用，而无法考虑到维护中产生的额外成本，所以维护团队中的 DBA 需要给开发团队的开发人员一定的培训，并且参与到各级评审中，尽早避免出现这种含有大量开关操作和占位操作的 SQL。

优化小技巧 优化计划调整

作为整体优化项目，必须有一个十分严格的实施计划，并且计划确定后，一定要严格按照计划实施。作为大型的优化项目，最大的忌讳就是经常调整优化计划。一般来说，优化计划一旦形成，就要尽可能按照计划实施，而不要随意提前。

在优化计划里，一般来说第一次实施的时间不要过早。任何对系统的变更都可能隐藏一部分系统的信息，导致我们无法正确掌握系统存在的实际问题，甚至可能把这些隐患遗留下来。在优化项目前期，系统出的问题越多越好，这样就可能将系统的主要问题全部抓住。

不过一般来说，来自客户那边的压力是很大的，因此经常会调整优化计划。最常碰到的事情

就是客户要求提前进行一些优化操作，以便于缓解系统存在的瓶颈。实际上这种要求是对优化项目危害最大的，但是很多情况下我们无法拒绝客户的要求，就像我们在沈阳的处境一样，客户实在是受不了了，我们必须提前做些什么。那么在这种情况下，我们尽可能做一些对后续分析影响较小的操作，尽可能不让临时性的优化操作掩盖更多的信息。下面几件是可以做的事情。

- 通过索引调整优化某些 SQL。
- 重建索引以提高索引的性能（通过 REBUILD 或者删除重建的方式重建索引）。
- 表整理：对于存在较多碎片的表进行重组以提高表访问的性能。
- 分析表和索引：对表和索引进行全面的分析以提高优化器的准确性。

优化小技巧　自动段存储空间管理 1

数据库空间管理是数据库中最重要的一个部分，也是 DBA 最重要的职责。DBA 会在数据库空间管理上花费大量的时间和精力。Oracle 9i 提供的新的数据库空间管理机制可以大大减轻 DBA 在这方面的工作。ASSM 就是这样一种新特性。在 Oracle 9i 之前的数据库版本中，段空间管理是通过 freelists 结构来实现的，DBA 在创建对象时可以定义 freelists 和 freelist_groups 的数量。pctused 参数决定了哪些数据块存放在 freelists 中。在 ASSM 机制中，段空间管理是通过位图来实现的。位图的状态指出每个块中空闲空间的状态（大于 75%、50%~75%、25%~50% 和小于 25%这 4 种状态）。在新的段空间管理模式下，freelists、freelist groups 和 pctused 的管理都不需要了，减轻了 DBA 的工作。并且在 ASSM 机制下，空闲空间的使用更加高效。在 ASSM 机制下，并发 DML 的性能也得到了改善，因为不同部分的位图可以被同步使用，而不像 freelists 那样，只能支持串行操作。

在 RAC 环境下，ASSM 也改善了 RAC 的性能。在 ASSM 机制下，各个节点不需要同步 freelists 和 freelist groups，节省了这方面的开销。Oracle 内部的基准测试结果表明，在一个双节点的 RAC 环境下，每台服务器配置 6 个 336 MHz 的 CPU 和 4GB 内存。在这个测试环境下，插入 300 条记录，使用 ASSM 机制比配置了 8 个 freelist groups、20 个 freelists 的手工模式快了大约 35%。

ASSM 仅仅在本地管理表空间中支持。在创建表空间时，加入 SEGMENT SPACE MANAGEMENT 子句允许 DBA 选择段空间管理模式（AUTO 或者 MANUAL）。下面的例子可以创建一个使用 ASSM 的表空间：

```
CREATE TABLESPACE data DATAFILE
'/u02/oracle/data/data01.dbf' SIZE 50M
 EXTENT MANAGEMENT LOCAL
 SEGMENT SPACE MANAGEMENT AUTO;
```

在这个表空间中创建的所有对象都会使用 ASSM 来管理空闲空间。在这种表空间中创建或修改对象的属性，所有的 pctused、freelists 和 freelist groups 参数都会被忽略。在 dba_tablespaces 视图中，增加了一个字段 segment_space_management 指明了该表空间是否使用 ASSM。要使用 ASSM，compatible 参数需要设置为 9.0.0 或者更高。

5月25日 倾斜的索引字段

今天早上刚想出门，小邓的电话打过来了。南京那边 ODS 的实验全部完成，除了发现当 Apply Handler 停止工作 3 个小时后，生产库出现了较为严重的共享池碎片，大量出现 ORA-4031 错误外，其他一切测试都很正常。针对 ORA-4031 的问题，通过加大共享池空间也得到了很大的缓解。通过一系列测试，我们的结论是：在 Oracle 9.2.0.8 环境下，Streams 工作一切正常，不过也发生过 Apply Handler 死掉的情况，后来编写了一个监控程序，如果发现 Apply Handler 死掉就自动重启，整个复制过程就很正常了。从上述试验来看，只要配以一定的监控手段，完全可以避免 Streams 给生产库造成严重影响的问题。我马上和李工通了电话，李工听到这个结果后也十分高兴。不过由于我们的测试环境是 Oracle 9.2.0.8，而且 Oracle 9.2.0.6 中存在几个 Streams 的 bug，因此我们还是建议客户升级到 Oracle 9.2.0.8。

打完电话，发现老于他们几个都已经去浑南找张工他们讨论 SQL 优化的安排了。今天的安排是我一个人在邮政大厦值班，其他人都在浑南总部，所以我马上打车过去。赶到办公室时已经 9 点多钟了，一进门，开发商的小刘就迎了上来，房间里还有几个开发人员，正围在那台监控电脑旁边讨论着什么，看见我进来，大家一下子围了上来。原来今天早上新上了一个业务模块，其中有一个应用涉及一张表的修改，要从一张表中将未处理的记录（`STATUS='A01P'`）取出来，经过处理后，再将状态位修改为'`A01F`'。只有少量的统计查询才需要查找 `A01F` 的数据，这种查询只占所有查询不到 1%。由于这张表一般情况下有 500 万左右的记录，其中 `A01P` 状态的记录大约在几百条到一千条，因此他们在这个字段上设计了一个索引，这个应用在测试时没有任何问题，但是今天一上线，就发现出现大量 ORA-60 错误。大约有 10%左右的业务失败，因此从 8：30 营业厅开业以来，营业前台的投诉就十分严重，甚至有些营业厅投诉到了公司老总那边。孙主任那边也感受到了很大的压力，要求开发商立即解决。

看到这个现象，我的第一感觉是，他们是不是使用了位图索引，因为位图索引锁住的不止是一行而是一组数据，所以出现死锁的可能性大大增加。我打开了一份 ORA-60 Trace 文件，查看了 SQL 语句和锁的情况，更加确定了我的怀疑，估计是位图索引导致的问题。于是我问了负责开发的小王，是不是他们在这张表上使用了位图索引。小王想了想，说确实是。我说位图索引一般用在 OLAP 系统上，在变化频率特别高的环境下不能使用位图索引，必须使用普通索引，否则在高并发量的情况下，会出现较多的死锁，前台会报 ORA-60 错误。

我这边正在和小王讨论如何优化，避免死锁的问题，旁边的一个技术人员立即删除了原有的

索引，并创建了一个 B 树索引。创建 B 树索引后，又出现了另外的问题，CPU 使用率一下子提高到 100%，整个系统出现了严重的性能问题。现场气氛一下子紧张了起来，现在是月底业务高峰期，也是投诉的高峰期，这个时候再出现这种事情，真是雪上加霜。通过 `v$session_wait` 可以看到系统出现了大量的 `db file scattered read`，看样子对那张表的访问开始走全表扫描了。

我立即要求开发商将那个出问题的 SQL 给我看看，我希望 STATUS 字段里面最好不使用绑定变量。如果没有使用绑定变量，那么分析柱状图后可以使用这个索引，这样就能够解决这个问题了。可不幸的是，STATUS 字段恰恰使用了绑定变量，这下子就很难办了，因为绑定窥视（bind peeking）可能会导致执行计划不稳定。因此，制订如下方案。

(1) 首先对该字段分析柱状图。

(2) 分析完毕立即执行如下代码：

```
ALTER SYSTEM FLUSH SHARED_POOL;
VAR PV VARCHAR2(10);
EXEC :PV:='A01P';
SELECT ......WHERE STATUS=:PV;
```

执行了上述方案后，CPU 使用率大幅度下降了，再检查那个 SQL 的执行计划，发现确实使用了新建的索引。大家悬着的心才逐渐放了下来，又继续观察了一段时间，在警告日志里已经看不到 ORA-60 的报错了，营业前台也没有新的投诉，看样子上面的操作是成功的。小王追着我问为什么要做第(2)步。关于绑定变量和绑定窥视，也不是一两句话就能说清楚的，我只好说这是 Oracle 的内部机制决定的，并且建议他们尽快修改应用程序，这个 SQL 不要使用绑定变量，因为今天的小手段只是一个临时性的措施，下次数据库重启或者其他情况下可能还会出问题。为了保证我的建议能够被完整地贯彻下去，我打了个电话给张工，让他督促一下小王他们，最好今天晚上就把应用改好部署上去。张工说，最好我能提交一个文档以我们和局方的名义一起提交给开发小组。我看了看目前的情况，也确实没有时间去写什么正规的文档了，看看小王还在办公室里，就直接又和他叮嘱了一番。为了让他更加重视，我有意夸大，告诉他今天的临时手段只能应付一两天，如果他们不修改程序，可能明后天还会出问题。小王听说后立即表态，今天晚上肯定修改完毕。DBA 这活确实不好干，为了达到目的，连撒谎的伎俩都用上了，我感觉脸上有些发烧，还好干这行时间长了，脸皮也厚了，小王应该是看不出来的。

今天是月底账期开始的时间，一般来说月底数据补录都是从 25 日开始的。所以今天系统的负载一直都很大，虽然刚才解决了那个全表扫描的问题，CPU 的使用率仍然很高，平均 CPU 等待队列一直在 100 以上。做了一个 cron 的作业，每隔 30 分钟采集一次 STATSPACK 采样。同时通过 OEM 开启了平均事务响应时间、Top SQL、闩锁分析和 I/O 成本分析这几个图表，并且对每个图表都开启了自动录像。

集成商的小孙今天一直很忙，不断有投诉电话打进来，不过小孙也没有什么办法。我打开了 BBS，发现今天的帖子倒不是很多，不过内容越来越尖锐了。由于系统的性能问题已经严重影响了营业部门的绩效考核，而绩效成绩决定了每个人的收入，所以就像孙主任说的，BBS 上就是骂

我们祖宗八代也是应该的。

下午系统还是那样，CPU 使用率基本上在 95%以上，不过偶尔还能看到一些 IDLE，通过 vmstat 看到的可运行队列一直在 50～100，小孙那边的电话也逐渐少了，看样子系统还算平稳。不过从上午的事情可以看出，这种在临界状态运行的系统，随时都有出事的可能，6 日的优化确实也是没办法的办法了，必须做点事情，而且必须对系统有较大的调整，否则这个系统真的很难说能不能撑过下个月的业务高峰。我又打开优化方案，认真地阅读起来，这回对优化方案真的是要多下一些工夫了。

下午老于他们回到了北站这边。我和老于讲了上午的事情，他说在总部那边就知道这边出事了，孙主任刚刚和他们讲了个开场白，就被叫了出去，然后就没有回来。后来听张工说出事了，本来都想暂停会议，让老于他们帮助处理一下，后来知道我已经到了现场并且在处理了，就没有动手。不过中午吃饭时我们在食堂里碰到孙主任，他也讲了这个事情，并且感谢我们处理得比较及时，帮助他们避免了一次大事故。

我和老于两个人坐在吸烟区一边喝茶一边抽烟，讨论着优化方案的问题，昨天我提出的这个优化范围是根据这几天的系统分析以及以往的经验做出的。一般来说，这些调整对系统的副作用是比较小的，肯定会使系统得到一定的缓解，不过由于这个系统的 CPU 负载相当高，而且服务器比较老，虽然当年购买的时候服务器的档次也不低，但现在看来配置相当低，其中有一套 P670 才配了 2 颗 CPU，作为一个省级运营商的服务器，确实有点力不从心。基于这个考虑，只要是对减少 CPU 有帮助的优化手段，我们都要考虑。

我和老于研究了一下 SQL 分析的问题，觉得如果能够减少 SQL 分析，也可以部分缓解 CPU 的使用率。实际上，我也很清楚，减少分析，顶多能减少 2%～3%的 CPU 使用，不过蚊子虽小也是肉啊。在这种情况下，只要能降低 CPU 使用率的方法就是好方法。通过分析应用，发现大量的 SQL 都没有使用绑定变量，如果将 cursor_sharing 调整为 similar 可以部分达到减少 SQL 分析的目的。

优化小技巧　倾斜字段的处理

在实际工作中，我们经常会碰到倾斜性的字段。比如像今天这个例子，在 500 万条记录的表中，STATUS 字段的取值只有区区几个。在这种情况下，大家可能想到的是使用位图索引，因为普通的 B 树索引由于键值取值范围太少，效率不高。不过位图索引由于其特殊的结构，操作某个记录时，会锁住某一组数据，而不是像普通索引那样仅锁住一行数据，因此位图索引往往在 OLAP 系统中使用，不适合并发访问十分高的 OLTP 系统，主要是因为位图索引会导致锁等待时间大幅度增加，甚至导致死锁频繁出现。

不过如果 STATUS 字段是倾斜的，就像今天的例子，50P 的值很少，只有几百条或者几千条记录，而大量的查询操作是要查出 STATUS='50P'的记录。对于倾斜的列，使用柱状图是常用的优化方案。这种技术使某列只有少量值时也能很好地使用 B 树索引。但是在这种情况下，最好不要使用绑定变量，否则由于绑定窥视，可能导致执行计划不稳定。这方面在 Oracle 10g 使用了新

的绑定窥视技术后有所改善。不过如果使用的是 Oracle 9i，建议还是在这种情况下不使用绑定变量。下面是一个实际的案例。

该系统存在严重的性能问题，I/O 十分高，DB Cache 的命中率不足 50%。经分析，散列值为 1285325803 的 SQL 存在严重的性能问题。通过 SPREPSQL 分析：

```
                                                          % Snap
                     Statement Total      Per Execute    Total
                     ---------------  ---------------  ------
           Buffer Gets:       1,055,004         36,379.4   32.81
            Disk Reads:       1,000,975         34,516.4   46.02
        Rows processed:             159              5.5
        CPU Time(s/ms):             113          3,889.3
    Elapsed Time(s/ms):             318         10,976.9
                 Sorts:              29              1.0
           Parse Calls:              29              1.0
         Invalidations:               0
         Version count:               1
       Sharable Mem(K):              49
            Executions:              29

SQL Text
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
SELECT        b.req_name, b.cli_pid, b.req_id, b.svr_name, b.timeout, b.svr_module,
  b.svr_inst, b.status, to_char(b.req_tm, 'YYYY MM DD HH24 MI SS'),
  nvl(a.default_timeout,0)
FROM service_list a, service_req_list b
WHERE
  a.app_id(+) = b.app_id and
  a.svr_name(+) = b.svr_name and
  b.status in ('Q','A') and
  b.app_id = :v1 order by b.req_tm,b.req_id

Known Optimizer Plan(s) for this Hash Value
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Shows all known Optimizer Plans for this database instance, and the Snap Id's
they were first found in the shared pool.  A Plan Hash Value will appear
multiple times if the cost has changed
-> ordered by Snap Id

 First        First          Plan
Snap Id     Snap Time     Hash Value        Cost
-------- --------------- ------------ ----------
     167 24 Sep 08 16:57     74934330     40682
Plans in shared pool between Begin and End Snap Ids
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Shows the Execution Plans found in the shared pool between the begin and end
snapshots specified.  The values for Rows, Bytes and Cost shown below are those
which existed at the time the first-ever snapshot captured this plan - these
values often change over time, and so may not be indicative of current values
-> Rows indicates Cardinality, PHV is Plan Hash Value
-> ordered by Plan Hash Value
```

```
------------------------------------------------------------------------------
| Operation                        | PHV/Object Name      |  Rows  | Bytes|   Cost  |
------------------------------------------------------------------------------
|SELECT STATEMENT                  |----- 74934330 ----- |        |      |  40682 |
|SORT ORDER BY                     |                      |     1M|  190M|  40682 |
| HASH JOIN OUTER                  |                      |     1M|  190M|   8355 |
|  TABLE ACCESS FULL               |SERVICE_REQ_LIST      |     1M|  147M|   5641 |
|  TABLE ACCESS BY INDEX ROWID     |SERVICE_LIST          |    15 |  345 |      2 |
|   INDEX RANGE SCAN               |PINX_SERVICE_LIST$_A  |     1 |      |      1 |
------------------------------------------------------------------------------
```

对 SERVICE_REQ_LIST 的全表扫描是引起性能问题的关键。该表有 100 多万条记录，其中存在下列过滤条件：

```
b.status in ('Q','A') and
b.app_id = :v1
```

经检查，b.app_id 的值域只有一个值，无选择性。而 b.status 只有很少的值，具体如下：

```
SQL> SELECT STATUS,COUNT(*) FROM acsdba.service_req_list GROUP BY status;
S   COUNT(*)
- ----------
A          2
C    2304679
E        367
Q          1
```

幸运的是，该 SQL 查找的 STATUS IN('Q','A')的记录只有几条。因此，STATUS 上的创建索引可以有效减少全表扫描。经检查发现，在 STATUS 字段上已经创建了索引，但是该索引并没有起作用，检查后发现是由于没有分析柱状图引起的。通过下列脚本重新分析该表：

```
exec DBMS_STATS.GATHER_TABLE_STATS(ownname=>'acsdba', tabname=>'service_req_list',
estimate_percent=>20, -
method_o> pt=>'for all indexed columns size skewonly',cascade=>true, degree=>2);
```

分析后检查执行计划，如下：

```
-------------------------------------------------------------------------------
| Operation                        | PHV/Object Name      |  Rows  | Bytes|   Cost  |
-------------------------------------------------------------------------------
|SELECT STATEMENT                  |----- 10932168 ----- |        |      |     25 |
|SORT ORDER BY                     |                      |     1  |  102 |     25 |
| NESTED LOOPS OUTER               |                      |     1  |  102 |      5 |
|  INLIST ITERATOR                 |                      |        |      |        |
|   TABLE ACCESS BY INDEX ROWID    |SERVICE_REQ_LIST      |     1  |   79 |      4 |
|    INDEX RANGE SCAN              |RINX_SERVICE_REQ$_LI  |     1  |      |      3 |
|  TABLE ACCESS BY INDEX ROWID     |SERVICE_LIST          |     1  |   23 |      1 |
|   INDEX UNIQUE SCAN              |UINX_SERVICE_LIST$_A  |     1  |      |        |
-------------------------------------------------------------------------------
```

发现全表扫描已经消除，通过 5 月 25 日上午的验证，系统的整体性能有了较大提升。

优化小技巧　自动段存储空间管理 2

下面通过一个例子来研究 Oracle 是如何通过位图来管理空闲空间的。首先创建一个表空间，

具体代码如下：

```
create tablespace t1
datafile '/data/ora9/test.dbf '
size 50m
EXTENT MANAGEMENT LOCAL
SEGMENT SPACE MANAGEMENT AUTO ;
```

通过查询 dba_tablespaces，可以确认表空间的属性具体如下所示：

```
SQL>select tablespace_name , extent_management , segment_space_management from
dba_tablespaces ;
TABLESPACE_NAME                EXTENT_MAN SEGMEN
------------------------------ ---------- ------
SYSTEM                         LOCAL      MANUAL
UNDOTBS1                       LOCAL      MANUAL
TEMP                           LOCAL      MANUAL
CWMLITE                        LOCAL      AUTO
DRSYS                          LOCAL      AUTO
EXAMPLE                        LOCAL      AUTO
ODM                            LOCAL      AUTO
TOOLS                          LOCAL      AUTO
USERS                          LOCAL      AUTO
XDB                            LOCAL      AUTO
CIQEMS                         LOCAL      MANUAL

TABLESPACE_NAME                EXTENT_MAN SEGMEN
------------------------------ ---------- ------
CREATEAM                       LOCAL      MANUAL
OUERLI                         LOCAL      MANUAL
TBS_DB                         LOCAL      MANUAL
T1                             LOCAL      AUTO

15 rows selected.
```

从上面可以看出，t1 表空间是 lmt 表空间，段空间管理是 AUTO 状态的。为了测试方便，下面建立两个账号，分别使用 t1 和 ciqems 作为默认表空间：

```
SQL> create user bmb_t1 identified by b default tablespace t1;
User created.
SQL> create user bmb_t2 identified by b default tablespace ciqems;
User created.
SQL> grant connect,resource to bmb_t1;
Grant succeeded.
SQL> grant connect ,resource to bmb_t2;
Grant succeeded.
```

在两个账号分别创建一张表，代码如下：

```
create table test_bmb (a integer,b varchar2(100));
```

并通过 PL/SQL 块插入数据，如下：

```
begin
  for b in 1..10 loop
       for i in 1..500 loop
```

```
            insert into test_bmb values (i,to_char(i*b*1234));
        end loop;
  end loop;
end;
/
```

在这两个账号中，分别对表进行分析：

```
analyze table test_bmb compute statistics;
```

通过系统用户连接数据库，然后查询：

```
SQL> SELECT owner,blocks, empty_blocks FROM dba_tables
  2  WHERE table_name='TEST_BMB';
OWNER                              BLOCKS EMPTY_BLOCKS
------------------------------ ---------- ------------
BMB_T2                                 12            3
BMB_T1                                 13            3
```

可以看出，同样的块大小，同样的数据，在不同段管理模式的表空间中出现了不一致的结果。使用 dbms_space.unused_space 存储过程去进行测试，结果如下：

```
SQL> set serveroutput on
SQL> declare
  2    TOTAL_BLOCKS              number;
  3    TOTAL_BYTES               number;
  4    UNUSED_BLOCKS             number;
  5    UNUSED_BYTES              number;
  6    LAST_USED_EXTENT_FILE_ID  number;
  7    LAST_USED_EXTENT_BLOCK_ID number;
  8    LAST_USED_BLOCK  number;
  9  begin
 10    dbms_space.unused_space('BMB_T1', 'TEST_BMB', 'TABLE',
 11       TOTAL_BLOCKS, TOTAL_BYTES, UNUSED_BLOCKS, UNUSED_BYTES,
 12       LAST_USED_EXTENT_FILE_ID, LAST_USED_EXTENT_BLOCK_ID,
 13       LAST_USED_BLOCK);
 14       dbms_output.put_line('OBJECT_NAME              = FREELIST_T');
 15       dbms_output.put_line('-----------------------------------');
 16       dbms_output.put_line('TOTAL_BLOCKS             = '||TOTAL_BLOCKS);
 17       dbms_output.put_line('UNUSED_BLOCKS            = '||UNUSED_BLOCKS);
 18       dbms_output.put_line('LAST_USED_BLOCK          = '||LAST_USED_BLOCK);
 19  end;
 20  /
OBJECT_NAME              = FREELIST_T
-----------------------------------
TOTAL_BLOCKS             = 16
UNUSED_BLOCKS            = 0
LAST_USED_BLOCK          = 8
PL/SQL 过程已成功完成。
SQL> declare
  2    TOTAL_BLOCKS              number;
  3    TOTAL_BYTES               number;
  4    UNUSED_BLOCKS             number;
  5    UNUSED_BYTES              number;
  6    LAST_USED_EXTENT_FILE_ID  number;
```

```
  7    LAST_USED_EXTENT_BLOCK_ID number;
  8    LAST_USED_BLOCK  number;
  9  begin
 10    dbms_space.unused_space('BMB_T2', 'TEST_BMB', 'TABLE',
 11       TOTAL_BLOCKS, TOTAL_BYTES, UNUSED_BLOCKS, UNUSED_BYTES,
 12       LAST_USED_EXTENT_FILE_ID, LAST_USED_EXTENT_BLOCK_ID,
 13       LAST_USED_BLOCK);
 14       dbms_output.put_line('OBJECT_NAME              = FREELIST_T');
 15       dbms_output.put_line('-----------------------------------');
 16       dbms_output.put_line('TOTAL_BLOCKS             = '||TOTAL_BLOCKS);
 17       dbms_output.put_line('UNUSED_BLOCKS            = '||UNUSED_BLOCKS);
 18       dbms_output.put_line('LAST_USED_BLOCK          = '||LAST_USED_BLOCK);
 19  end;
 20  /
OBJECT_NAME              = FREELIST_T
-----------------------------------
TOTAL_BLOCKS             = 16
UNUSED_BLOCKS            = 3
LAST_USED_BLOCK          = 5
PL/SQL 过程已成功完成。
```

从上面的例子可以看出，使用两种段空间管理模式时，数据存储方式也不同。使用 ASSM，16 个数据块中都存储了数据，而使用手工管理模式时，有 3 个块是在 freelists 上的。对于手工管理模式，总计占用了 16 个块，其中一个块存储段头信息，15 个块存储数据，因此从 DBA_TABLES 中会看到少了一块。由于 ASSM 模式改变了空闲块的概念，所以使用 dbms_space.unused_space 得到的信息不能说明问题。

通过一个新的存储过程可以查看 ASSM 表空间中的空间使用情况，这个存储过程就是 dbms_space.space_usage。下面用这个存储过程来检查 TEST_BMB 的空间使用情况：

```
SQL> declare
  2     v_unformatted_blocks number;
  3     v_unformatted_bytes number;
  4     v_fs1_blocks number;
  5     v_fs1_bytes number;
  6     v_fs2_blocks number;
  7     v_fs2_bytes number;
  8     v_fs3_blocks number;
  9     v_fs3_bytes number;
 10     v_fs4_blocks number;
 11     v_fs4_bytes number;
 12     v_full_blocks number;
 13     v_full_bytes number;
 14   begin
 15     dbms_space.space_usage ('BMB_T1', 'TEST_BMB', 'TABLE',
 16       v_unformatted_blocks,
 17       v_unformatted_bytes, v_fs1_blocks, v_fs1_bytes, v_fs2_blocks, v_fs2_by
          tes,
 18       v_fs3_blocks, v_fs3_bytes, v_fs4_blocks, v_fs4_bytes, v_full_blocks, v
          _full_bytes);
 19     dbms_output.put_line('Unformatted Blocks = '||v_unformatted_blocks);
```

```
 20     dbms_output.put_line('FS1 Blocks              = '||v_fs1_blocks);
 21     dbms_output.put_line('FS2 Blocks              = '||v_fs2_blocks);
 22     dbms_output.put_line('FS3 Blocks              = '||v_fs3_blocks);
 23     dbms_output.put_line('FS4 Blocks              = '||v_fs4_blocks);
 24     dbms_output.put_line('Full Blocks             = '||v_full_blocks);
 25   end;
 26  /
Unformatted Blocks = 0
FS1 Blocks         = 0
FS2 Blocks         = 0
FS3 Blocks         = 0
FS4 Blocks         = 2
Full Blocks        = 11

PL/SQL 过程已成功完成。
```

可以看出，这个结果与表分析后从 DBA_TABLES 中获得的结果一致。目前系统中有 11 个满的块，2 个超过 75%的块，3 个空块。

在上面的结果中，FS1～FS4 的定义如下。

- FS1：0%～25%　空间使用率。
- FS2：25%～50% 空间使用率 。
- FS3：50%～75% 空间使用率 。
- FS4：75%～100% 空间使用率 。

为什么会有 3 个空块呢？下面我们就来揭开这 3 个块的秘密。为了更有针对性，我们继续通过 PL/SQL 块来增加数据，然后通过 dba_extents 视图查看扩展（EXTENT）的情况：

```
SQL> COL SEGMENT_NAME FORMAT A30 TRUNCATE;
SQL> SELECT segment_name,extent_id,file_id,block_id
  2  FROM dba_extents  WHERE segment_name = 'TEST_BMB' AND OWNER='BMB_T1';
SEGMENT_NAME                    EXTENT_ID    FILE_ID   BLOCK_ID
------------------------------ ---------- ---------- ----------
TEST_BMB                                0         14          9
TEST_BMB                                1         14         17
TEST_BMB                                2         14         25
TEST_BMB                                3         14         33
TEST_BMB                                4         14         41
TEST_BMB                                5         14         49
TEST_BMB                                6         14         57
TEST_BMB                                7         14         65
TEST_BMB                                8         14         73
TEST_BMB                                9         14         81
TEST_BMB                               10         14         89

SEGMENT_NAME                    EXTENT_ID    FILE_ID   BLOCK_ID
------------------------------ ---------- ---------- ----------
TEST_BMB                               11         14         97
TEST_BMB                               12         14        105
TEST_BMB                               13         14        113
TEST_BMB                               14         14        121
TEST_BMB                               15         14        129
```

```
TEST_BMB                                    16          14          137
TEST_BMB                                    17          14          265
```

可以看出，TEST_BMB 有 18 个扩展,第一个是从 14 号文件的 BLOCK_ID=9 开始的，第二个是从 14 号文件的 BLOCK_ID=17 开始的。下面我们来分析 14 号文件的 9、10、11 这 3 个块：

```
alter system dump datafile 14 block 9;
alter system dump datafile 14 block 10;
alter system dump datafile 14 block 11;
```

下面是转储的内容：

```
Start dump data blocks tsn: 22 file#: 14 minblk 9 maxblk 9
buffer tsn: 22 rdba: 0x03800009 (14/9)
scn: 0x0000.00f8ecb8 seq: 0x02 flg: 0x04 tail: 0xecb82002
frmt: 0x02 chkval: 0x26cf type: 0x20=FIRST LEVEL BITMAP BLOCK
Dump of First Level Bitmap Block
 --------------------------------
   nbits : 4 nranges: 2         parent dba:  0x0380000a   poffset: 0
   unformatted: 0       total: 16        first useful block: 3
   owning instance : 1
   instance ownership changed at 12/01/2004 12:10:26
   Last successful Search 12/01/2004 12:10:26
   Freeness Status:  nf1 0      nf2 0      nf3 0      nf4 0

   Extent Map Block Offset: 4294967295
   First free datablock : 16
   Bitmap block lock opcode 0
   Locker xid:     :  0x0000.000.00000000
      Highwater::  0x00000000  ext#: 0      blk#: 0      ext size: 0
  #blocks in seg. hdr's freelists: 0
  #blocks below: 0
  mapblk  0x00000000  offset: 0
  HWM Flag: Not Set
  --------------------------------------------------------
  DBA Ranges :
  --------------------------------------------------------
   0x03800009  Length: 8      Offset: 0
   0x03800011  Length: 8      Offset: 8

   0:Metadata   1:Metadata   2:Metadata   3:FULL
   4:FULL   5:FULL   6:FULL   7:FULL
   8:FULL   9:FULL   10:FULL   11:FULL
   12:FULL   13:FULL   14:FULL   15:FULL
  --------------------------------------------------------
End dump data blocks tsn: 22 file#: 14 minblk 9 maxblk 9
Start dump data blocks tsn: 22 file#: 14 minblk 10 maxblk 10
buffer tsn: 22 rdba: 0x0380000a (14/10)
scn: 0x0000.00f8ee7c seq: 0x01 flg: 0x00 tail: 0xee7c2101
frmt: 0x02 chkval: 0x0000 type: 0x21=SECOND LEVEL BITMAP BLOCK
Dump of Second Level Bitmap Block
   number: 12      nfree: 2       ffree: 10     pdba:     0x0380000b
  opcode:0
 xid:
```

```
  L1 Ranges :
  --------------------------------------------------------
   0x03800009  Free: 1 Inst: 1
   0x03800019  Free: 1 Inst: 1
   0x03800029  Free: 1 Inst: 1
   0x03800039  Free: 1 Inst: 1
   0x03800049  Free: 1 Inst: 1
   0x03800059  Free: 1 Inst: 1
   0x03800069  Free: 1 Inst: 1
   0x03800079  Free: 1 Inst: 1
   0x03800089  Free: 1 Inst: 1
   0x0380008a  Free: 1 Inst: 1
   0x03800109  Free: 5 Inst: 1
   0x0380010a  Free: 5 Inst: 1

  --------------------------------------------------------
End dump data blocks tsn: 22 file#: 14 minblk 10 maxblk 10
Start dump data blocks tsn: 22 file#: 14 minblk 11 maxblk 11
buffer tsn: 22 rdba: 0x0380000b (14/11)
scn: 0x0000.00f8ee7c seq: 0x02 flg: 0x00 tail: 0xee7c2302
frmt: 0x02 chkval: 0x0000 type: 0x23=PAGETABLE SEGMENT HEADER
  Extent Control Header
  -----------------------------------------------------------------
  Extent Header:: spare1: 0      spare2: 0      #extents: 18     #blocks: 384
                  last map  0x00000000  #maps: 0      offset: 2716
      Highwater::  0x03800149  ext#: 17     blk#: 64     ext size: 128
  #blocks in seg. hdr's freelists: 0
  #blocks below: 306
  mapblk  0x00000000  offset: 17
                   Unlocked
  --------------------------------------------------------
  Low HighWater Mark :
      Highwater::  0x03800089  ext#: 15     blk#: 8      ext size: 8
  #blocks in seg. hdr's freelists: 0
  #blocks below: 118
  mapblk  0x00000000  offset: 15
  Level 1 BMB for High HWM block: 0x03800109
  Level 1 BMB for Low HWM block: 0x03800079
  --------------------------------------------------------
  Segment Type: 1 nl2: 1      blksz: 8192   fbsz: 0
  L2 Array start offset:  0x00001434
  First Level 3 BMB:  0x00000000
  L2 Hint for inserts:  0x0380000a
  Last Level 1 BMB:  0x0380010a
  Last Level II BMB:  0x0380000a
  Last Level III BMB:  0x00000000
     Map Header:: next  0x00000000  #extents: 18   obj#: 34180  flag: 0x20000000
  Extent Map
  -----------------------------------------------------------------
   0x03800009  length: 8
   0x03800011  length: 8
   0x03800019  length: 8
   0x03800021  length: 8
   0x03800029  length: 8
   0x03800031  length: 8
```

```
   0x03800039  length: 8
   0x03800041  length: 8
   0x03800049  length: 8
   0x03800051  length: 8
   0x03800059  length: 8
   0x03800061  length: 8
   0x03800069  length: 8
   0x03800071  length: 8
   0x03800079  length: 8
   0x03800081  length: 8
   0x03800089  length: 128
   0x03800109  length: 128
    Auxillary Map
  --------------------------------------------------------
   Extent 0     :  L1 dba:  0x03800009 Data dba:  0x0380000c
   Extent 1     :  L1 dba:  0x03800009 Data dba:  0x03800011
   Extent 2     :  L1 dba:  0x03800019 Data dba:  0x0380001a
   Extent 3     :  L1 dba:  0x03800019 Data dba:  0x03800021
   Extent 4     :  L1 dba:  0x03800029 Data dba:  0x0380002a
   Extent 5     :  L1 dba:  0x03800029 Data dba:  0x03800031
   Extent 6     :  L1 dba:  0x03800039 Data dba:  0x0380003a
   Extent 7     :  L1 dba:  0x03800039 Data dba:  0x03800041
   Extent 8     :  L1 dba:  0x03800049 Data dba:  0x0380004a
   Extent 9     :  L1 dba:  0x03800049 Data dba:  0x03800051
   Extent 10    :  L1 dba:  0x03800059 Data dba:  0x0380005a
   Extent 11    :  L1 dba:  0x03800059 Data dba:  0x03800061
   Extent 12    :  L1 dba:  0x03800069 Data dba:  0x0380006a
   Extent 13    :  L1 dba:  0x03800069 Data dba:  0x03800071
   Extent 14    :  L1 dba:  0x03800079 Data dba:  0x0380007a
   Extent 15    :  L1 dba:  0x03800079 Data dba:  0x03800081
   Extent 16    :  L1 dba:  0x03800089 Data dba:  0x0380008b
   Extent 17    :  L1 dba:  0x03800109 Data dba:  0x0380010b
  --------------------------------------------------------
     Second Level Bitmap block DBAs
   --------------------------------------------------------
   DBA 1:   0x0380000a
  End dump data blocks tsn: 22 file#: 14 minblk 11 maxblk 11
```

可以看出第一个块(9 号块)保存了一个 L1 BMB(一级位图块),第二个块保存了一个 L2 BMB(二级位图块)，第三个块是段头信息。

Oracle 位图的结构是：每个 L1 BMB 包含 16 个数据块的位图信息，如下所示：

```
0:Metadata   1:Metadata   2:Metadata   3:FULL
   4:FULL   5:FULL   6:FULL   7:FULL
   8:FULL   9:FULL   10:FULL   11:FULL
   12:FULL   13:FULL   14:FULL   15:FULL
```

在 L2 BMB 中存放了 L1 BMB 的地址，而在 L3 BMB 中存放了 L2 BMB 的地址。在段头块（第三个块）中，存放了 L3 BMB 和 L2 BMB 的相关信息。如下所示：

```
Segment Type: 1 nl2: 1      blksz: 8192   fbsz: 0
  L2 Array start offset:  0x00001434
  First Level 3 BMB:  0x00000000
  L2 Hint for inserts:  0x0380000a
```

```
    Last Level 1 BMB:  0x0380010a
    Last Level II BMB:  0x0380000a
    Last Level III BMB:  0x00000000
```

在这部分信息中，包含了第一个 L3 BMB 的地址（这里是 0，表示没有 L3 BMB）以及最后一个各级 BMB 的地址。通过这个信息，Oracle 数据库可以掌握所有位图块的地址信息。事情看起来好像很简单，所有的疑问都得到了解释。但是为什么一个 L1 BMB 只存放 16 个块的位图呢？为什么不存放 64 个块的位图呢？如果使用 32KB 的块大小，那么不是很浪费吗？在段头信息中有一张扩展表的数据引起了我们的注意，如下：

```
  Extent Map
    -----------------------------------------------------------------
     0x03800009  length: 8
     0x03800011  length: 8
     0x03800019  length: 8
     0x03800021  length: 8
     0x03800029  length: 8
     0x03800031  length: 8
     0x03800039  length: 8
     0x03800041  length: 8
     0x03800049  length: 8
     0x03800051  length: 8
     0x03800059  length: 8
     0x03800061  length: 8
     0x03800069  length: 8
     0x03800071  length: 8
     0x03800079  length: 8
     0x03800081  length: 8
     0x03800089  length: 128
     0x03800109  length: 128
      Auxillary Map
    --------------------------------------------------------
     Extent 0     :  L1 dba:  0x03800009 Data dba:  0x0380000c
     Extent 1     :  L1 dba:  0x03800009 Data dba:  0x03800011
     Extent 2     :  L1 dba:  0x03800019 Data dba:  0x0380001a
     Extent 3     :  L1 dba:  0x03800019 Data dba:  0x03800021
     Extent 4     :  L1 dba:  0x03800029 Data dba:  0x0380002a
     Extent 5     :  L1 dba:  0x03800029 Data dba:  0x03800031
     Extent 6     :  L1 dba:  0x03800039 Data dba:  0x0380003a
     Extent 7     :  L1 dba:  0x03800039 Data dba:  0x03800041
     Extent 8     :  L1 dba:  0x03800049 Data dba:  0x0380004a
     Extent 9     :  L1 dba:  0x03800049 Data dba:  0x03800051
     Extent 10    :  L1 dba:  0x03800059 Data dba:  0x0380005a
     Extent 11    :  L1 dba:  0x03800059 Data dba:  0x03800061
     Extent 12    :  L1 dba:  0x03800069 Data dba:  0x0380006a
     Extent 13    :  L1 dba:  0x03800069 Data dba:  0x03800071
     Extent 14    :  L1 dba:  0x03800079 Data dba:  0x0380007a
     Extent 15    :  L1 dba:  0x03800079 Data dba:  0x03800081
     Extent 16    :  L1 dba:  0x03800089 Data dba:  0x0380008b
     Extent 17    :  L1 dba:  0x03800109 Data dba:  0x0380010b
    --------------------------------------------------------
```

从第 17 次扩展开始，length 变成了 128。通过 Auxillary Map 也可以看到，从第 17 次扩展开始，每次扩展不是 8 块，而是 128 块。由于 T1 表空间设置为 AUTOALLOCATE，所以默认的扩展大小为 1MB，表示 BMB_TEST 的 INITIAL=NEXT=64KB，当 BMB_TEST 扩展 16 次后，总的空间超过 1MB，因此下次扩展时，系统自动使用 1MB 进行扩展。如果表的 INITIAL 和 NEXT 设置超过表空间统一扩展大小时，系统按照表空间统一扩展大小来扩展空间，而不是按照表设定的值来扩展空间，所以从第 17 次扩展开始，length 变成了 128 块（1MB）。

另外，从这张表中可以清楚地看到每次扩展的位图信息存放的地址以及数据开始块的地址。值得注意的是，对于 Extent 2，L1 BMB 的地址是 0x19，数据是 0x1a，就是下一快。而 Extent 16 的 L1 BMB 的地址是 0x89，而数据是 0x8b，而不是我们推算的 0x9a。为什么会出现这种情况呢？0x8a 中又会有什么新的发现呢？下面就是 0x89 和 0x8a 的数据。

```
ALTER SYSTEM DUMP DATAFILE 14 block 137;
ALTER SYSTEM DUMP DATAFILE 14 block 138;
```

转储的结果如下：

```
Start dump data blocks tsn: 22 file#: 14 minblk 137 maxblk 137
buffer tsn: 22 rdba: 0x03800089 (14/137)
scn: 0x0000.00f8ee1e seq: 0x04 flg: 0x00 tail: 0xee1e2004
frmt: 0x02 chkval: 0x0000 type: 0x20=FIRST LEVEL BITMAP BLOCK
Dump of First Level Bitmap Block
 --------------------------------
   nbits : 4 nranges: 1         parent dba:  0x0380000a   poffset: 8
   unformatted: 0       total: 64        first useful block: 2
   owning instance : 1
   instance ownership changed at 12/01/2004 13:17:49
   Last successful Search 12/01/2004 13:17:49
   Freeness Status:  nf1 0      nf2 0      nf3 0      nf4 0

   Extent Map Block Offset: 4294967295
   First free datablock : 64
   Bitmap block lock opcode 0
   Locker xid:     :  0x0000.000.00000000
      Highwater::  0x00000000  ext#: 0      blk#: 0      ext size: 0
  #blocks in seg. hdr's freelists: 0
  #blocks below: 0
  mapblk  0x00000000  offset: 0
  HWM Flag: Not Set
  --------------------------------------------------------
  DBA Ranges :
  --------------------------------------------------------
   0x03800089  Length: 64     Offset: 0

   0:Metadata   1:Metadata   2:FULL   3:FULL
   4:FULL   5:FULL   6:FULL   7:FULL
   8:FULL   9:FULL   10:FULL   11:FULL
   12:FULL   13:FULL   14:FULL   15:FULL
   16:FULL   17:FULL   18:FULL   19:FULL
   20:FULL   21:FULL   22:FULL   23:FULL
```

```
   24:FULL   25:FULL   26:FULL   27:FULL
   28:FULL   29:FULL   30:FULL   31:FULL
   32:FULL   33:FULL   34:FULL   35:FULL
   36:FULL   37:FULL   38:FULL   39:FULL
   40:FULL   41:FULL   42:FULL   43:FULL
   44:FULL   45:FULL   46:FULL   47:FULL
   48:FULL   49:FULL   50:FULL   51:FULL
   52:FULL   53:FULL   54:FULL   55:FULL
   56:FULL   57:FULL   58:FULL   59:FULL
   60:FULL   61:FULL   62:FULL   63:FULL
  --------------------------------------------------------
End dump data blocks tsn: 22 file#: 14 minblk 137 maxblk 137
*** 2004-12-01 13:44:21.979
Start dump data blocks tsn: 22 file#: 14 minblk 138 maxblk 138
buffer tsn: 22 rdba: 0x0380008a (14/138)
scn: 0x0000.00f8ee7c seq: 0x02 flg: 0x00 tail: 0xee7c2002
frmt: 0x02 chkval: 0x0000 type: 0x20=FIRST LEVEL BITMAP BLOCK
Dump of First Level Bitmap Block
 --------------------------------
   nbits : 4 nranges: 1         parent dba:  0x0380000a   poffset: 9
   unformatted: 0       total: 64        first useful block: 0
   owning instance : 1
   instance ownership changed at 12/01/2004 13:17:56
   Last successful Search 12/01/2004 13:17:56
   Freeness Status:  nf1 0      nf2 0      nf3 0      nf4 0

   Extent Map Block Offset: 4294967295
   First free datablock : 64
   Bitmap block lock opcode 0
   Locker xid:     :  0x0000.000.00000000
      Highwater::  0x00000000  ext#: 0      blk#: 0      ext size: 0
  #blocks in seg. hdr's freelists: 0
  #blocks below: 0
  mapblk  0x00000000  offset: 0
  HWM Flag: Not Set
  --------------------------------------------------------
  DBA Ranges :
  --------------------------------------------------------
   0x038000c9  Length: 64     Offset: 0

   0:FULL   1:FULL   2:FULL   3:FULL
   4:FULL   5:FULL   6:FULL   7:FULL
   8:FULL   9:FULL   10:FULL   11:FULL
   12:FULL   13:FULL   14:FULL   15:FULL
   16:FULL   17:FULL   18:FULL   19:FULL
   20:FULL   21:FULL   22:FULL   23:FULL
   24:FULL   25:FULL   26:FULL   27:FULL
   28:FULL   29:FULL   30:FULL   31:FULL
   32:FULL   33:FULL   34:FULL   35:FULL
   36:FULL   37:FULL   38:FULL   39:FULL
   40:FULL   41:FULL   42:FULL   43:FULL
   44:FULL   45:FULL   46:FULL   47:FULL
   48:FULL   49:FULL   50:FULL   51:FULL
   52:FULL   53:FULL   54:FULL   55:FULL
```

```
   56:FULL   57:FULL   58:FULL   59:FULL
   60:FULL   61:FULL   62:FULL   63:FULL
  --------------------------------------------------------
End dump data blocks tsn: 22 file#: 14 minblk 138 maxblk 138
```

原来这两个数据块都是 L1 BMB。在 0x89 块中可以看到，这个位图中有 2 个元数据块，就是 0x89/0x8a。在这两个块中存放了 128 个块的位图信息。从上面的信息我们可以进行如下的推断。

- L1 BMB 不是固定的 1 块，而是根据每次扩展的块的数量来确定。
- 每组 L1 BMB 至少能够存储 1 次扩展的数据的位图。
- 每个 L1 BMB 至少存储 16 个块的位图。
- 每个 L1 BMB 最多存储 64 个块的位图。

下面我们通过一系列实验来证明这个推断。首先创建一张表，设置 INITIAL 为 2MB，然后对段头进行分析。

为了便于分析，我们创建了一个新的表空间 T2。

```
CREATE TABLESPACE T2
    DATAFILE '/data/ora9/T2.dbf' SIZE 50M
    EXTENT MANAGEMENT LOCAL
    UNIFORM SIZE 2048K
    SEGMENT SPACE MANAGEMENT  AUTO;
```

在 BMB_T1 用户下，执行下列脚本，创建一张测试表：

```
DROP table test_bmb;
CREATE table test_bmb (a integer,b varchar2(100))
storage(initial 2M) tablespace t2;
```

在系统用户下，可以通过 daa_extents 视图查询该表的扩展信息，从中找到文件号（file_id）和块号（block_id）：

```
SQL> COL SEGMENT_NAME FORMAT A30 TRUNCATE;
SQL> SELECT segment_name,extent_id,file_id,block_id
  2  FROM dba_extents  WHERE
  3  segment_name = 'TEST_BMB' AND OWNER='BMB_T1';
SEGMENT_NAME                   EXTENT_ID    FILE_ID   BLOCK_ID
------------------------------ ---------- ---------- ----------
TEST_BMB                                0         15          9
```

将 15 号文件 9～14 块转储出来，可以看出一共有 6 个元数据块，其中 L1 BMB 4 块、L2 BMB 1 块以及段头 2 块。从上面的例子可以证明前面的推断，转储相关数据的脚本如下：

```
ALTER SYSTEM DUMP DATA FILE15 block min9 block max 14;
```

5月26日 优化方案

6 月 6 日要进行第一次优化实施，现在我们的时间也不多了。这几天必须继续监控系统，因为这几天是月底业务高峰期，大量的补单操作会给系统负载带来较大的影响，而这几天的观察可能会带来一些新的发现。除了监控以外，还有一件十分紧迫的事情，就是必须马上做出一个切实可行的优化方案。优化方案还要经过开发商的审核，并且和甲方一起确认，这都需要时间，我们在后面的 10 天里能够完成这些工作吗？如果要在 6 月 6 日顺利实施第一批优化，我们必须在这个星期完成整个第一期优化方案。并且为了配合优化实施，在 6 月 6 日实施之前，必须优化更多的 SQL，使 CPU 开销尽可能多地下降。看样子老熊这几天就不能参与优化方案设计了，他必须加快优化一些 SQL。开发商修改应用的难度较大，老熊这几天优化的重点也放在了能够通过添加索引进行优化的 SQL 上面。

由于昨天的突发事件浪费了一天的宝贵时间，所以今天必须将优化方案的框架确定下来，然后分工准备。老熊正在紧张地分析 SQL，老肖也在分析热点对象。我和老于坐在吸烟区，每个人抱着一大杯茶水，叼着香烟，讨论着初步的优化方案。根据目前对系统的分析结果，我们必须在 6 月 6 日解决以下几个问题。

(1) 需要解决系统物理内存不足的问题。从目前情况来看，I/O 系统的负载过高，存储子系统的响应速度是正常值的几倍。因为系统较为复杂，存储系统比较老，并且容量也存在不足，所以基本没有从存储系统本身解决问题的可能性，而最有效的应用优化工作又进展缓慢。除了应用优化和存储优化之外，解决 I/O 问题的方法是要增加 DB Cache 的大小，因此必须从操作系统获取更多的物理内存。通过前一段时间的分析，大约有 30%～40%的物理内存被操作系统文件缓冲占用，这是因为 `maxperm%`参数使用了默认的 80%。通过调整 `maxperm%`参数，可以让操作系统释放大量的物理内存，从而为 DB Cache 扩容提供有利的条件。我们建议将 `maxperm%`调整为 10%，`minperm%`调整为 5%。

(2) 为了解决目前的 I/O 性能问题，必须加大 DB Cache，同时，由于系统总的物理内存十分有限，除了加大 default 池的大小外，为了能够提高 DB Cache 的性能，我们建议客户启用 keep 池。启用 keep 池是提高 DB Cache 性能、减小 I/O 数量的最常用的办法。在我以前做过的优化案例中，每次使用这个方法都取得了不错的效果。之前老肖已经分析了一些表和索引，准备了一个可以放入 keep 池对象的清单。我们可以通过 `ALTER` 命令修改表和索引的属性，把常用的对象放到 keep 池中。

(3) 加大 DB Cache 后势必增加 CPU 的开销，因此必须采取一定的措施将 CPU 的使用率降下来。减少 CPU 开销的最佳方法是 SQL 优化，但是从开发商目前的进展来看，在 6 月 6 日提交一些成果的可能性很小，因此我们决定从两方面入手：一方面是对几张大表进行分区，这几张大表产生的逻辑读和物理读占到整个系统的 80%以上，通过分区，可以减少大量不必要的逻辑读和物理读；另一方面是重新设计对系统性能影响较大的 SQL 相关表的索引，提高整个系统的效率，减少缓冲区获取的数量，这部分工作由老熊承担。还有一个屡试不爽的减少 CPU 开销的办法是把所有的索引都重建一遍，这样可以消除索引碎片，减少索引大小，从而提高索引使用的效率。前几天我分析索引时，居然发现有一张表本身只有 30 多兆字节，主键索引的大小就超过 50MB，这种索引如果重建一下，估计可能缩小到几兆字节。另外就是对于访问比较频繁的中等大小和较小的表，如果这些表存在较多的碎片，那么通过碎片整理，也可以减少缓冲区获取的数量，从而降低 CPU 的使用率。

(4) 调整共享池以及相关的参数，减少硬分析的消耗。由于本系统硬分析很严重，大概消耗了 5%～10%的 CPU 资源，所以调整共享池可以少量减少 CPU 的消耗，同时提高系统效率。由于本系统中绝大多数 SQL 没有使用绑定变量，所以在共享池优化方面，我们准备将 cursor_sharing 从目前的 exact 调整为 similar，另外准备启用 session_cached_cursors（目前这个参数为 0，这个时候会话的游标缓冲未被启用），将这个参数设置为 100。

(5) 对所有的表和索引进行一次 30%采样的全面分析，解决目前部分 SQL 执行计划不正确的问题。虽然 30%采样比较大，对于绝大多数表来说，10%采样就能够保证执行计划的准确性了。但是我们一方面缺乏测试环境来测试最小采样比例，另一方面时间紧迫也没办法逐一测试，所以只能选择这个相对比较安全的采样比例。

(6) 还必须注意排队效应。排队效应是优化工作中经常遇到的一个问题。如果针对一个没有瓶颈的系统或者是一个封闭的“黑匣子”，是不会出现排队效应的。举个例子，如果在某个单位时间内系统负载是不变的，那么任何一个优化动作都可能会带来好处。而对于一般的生产系统，由于资源瓶颈出现性能问题的系统，往往会产生大量的业务积压，也就是排队。当系统性能提升时，这些积压的排队事务就会涌入，导致系统负载上升，而当负载上升到某一个阈值时，系统资源将会出现新的瓶颈，从而导致系统性能恶化，将优化的成果完全消耗殆尽，甚至将性能拉回优化前的状态。因此，我们在第一批实施时，必须考虑排队效应，我们做的优化必须足以抵消排队效应带来的负面影响，否则本次优化注定要失败。因此，我们 6 月 6 日实施的力度要足够大，最关键的是要大幅度减少 CPU 的开销。

我和老于两个人一边喷云吐雾一边核对这个优化方案。从目前来看，这个实施方案应该是很周密和可行的。两个人反复对着写着草稿的纸看了几遍，感觉应该没有什么问题了。下一步就要看明天的三方沟通会上开发厂商能否支持这个方案了。如果开发商不支持这个优化方案，那么由于目前 CPU 存在瓶颈，我们这个优化的成败就取决于表分区、索引优化和表分析这三招的效果了。如果这三个手段不能将缓冲区获取降低 20%以上，那么这个方案就存在一定的风险了。如果由于排队效应，导致优化后 CPU 出现瓶颈，那么这次调整可能会起不到应有的效果，甚至可能

出现系统性能不升反降的不利局面。如果真的出现了这种情况，我们能否完成这个项目都成问题了。优化项目做到一半就被客户赶回家的案例还是挺多的。

我刚把这个疑虑说出来，老于也皱了皱眉，最后无奈地说："爱谁谁吧，如果开发商不配合，就只能请集团公司死压他们了。"连老于这样的老实人都说出"爱谁谁"的话来，开发商真是把我们逼得够惨了。目前这个优化方案的关键点就是表分区、索引优化、表碎片整理等操作能否使 CPU 的使用率下降 20%。说实在的，我还是比较担心排队效应，实际上我们并不清楚营业厅的实际情况，如果优化实施后，平均每秒的事务数量提高一至两倍都是可能的。如果 SQL 优化不同步进行，那么优化后如果平均事务数量提高了一倍以上，则由于排队事务的涌入，可能导致系统负载出现超过 30%以上的增加，此时我们通过优化节省下来的 20%的 CPU 容量很可能被新涌入的事务完全消耗光。这样优化的效果就很难保证了。可能会出现忙了半天系统却没有什么改善的情况。如果发生这种情况，现在十分支持我们的孙主任还会继续支持我们吗？前几天小齐打电话来说另外一个试点——山西那边的优化项目陷入了困境，我们的竞争对手做了两次优化，都没有达到预定的效果，反而出了几次系统故障。山西的客户已经决定停止优化项目，宣布优化失败了。山西的情况给我们敲响了警钟，我们千万不能重蹈覆辙。

优化小技巧 优化中的排队效应

今天的日记中提到了一个比较陌生的名词"排队效应"。好像这也不是一个正规的术语，是我自己在优化实践中发现的，用了一个比较形象的词来形容。刚刚开始从事优化工作的 DBA 往往会陷入一个误区，就是认为系统的负载是基本稳定的，在这种前提下，做任何一项优化工作都是积极的，很少会产生负面效应。而事实却截然相反，经常会出现做了优化反而性能下降的情况。其实我们优化时经常会发现，优化后平均事务响应时间会有所提升，每秒处理的事务更多了，每秒产生的重做量更大了，缓冲区获取（buffer get）、缓冲区写（buffer write）也相应增加了。对于各种系统资源较为充裕的系统，排队效应对优化的影响不会很大，系统优化后，各种资源得到了充分的发挥，虽然系统的 CPU、内存和 I/O 资源的使用率都有所增加，但并不会出现资源瓶颈。可是对于存在较为严重资源瓶颈的系统，排队效应需要被充分考虑。

实际上，当某个影响系统性能的等待事件减少了，平均每秒处理的事务量就增加了，忙时集中系数（忙时集中系数最早被电信运营商使用，其含义是最忙的 1 小时内处理的业务量占全天 24 小时的百分比，比如忙时集中系数是 0.1，那么最忙的一个小时处理的业务量是全天平均值的 2.4 倍）增加了，那么对于系统资源的消耗肯定会有所增长。如果某个资源存在瓶颈，那么这种优化可能导致存在瓶颈的资源消耗增大，如果该资源的消耗增大到出现过载的情况，系统性能会急剧下降，使系统的整体性能不会提升反而会下降。

上面说的是系统中的排队效应，是一种系统内部的逻辑排队。对于存在营业前台的营业系统，还有一个实际物理上的排队。如果优化前的平均事务响应时间是 3s，那么一个营业员可能每分钟可以处理 2 笔业务，营业窗口可能排起几米长的队，而如果系统响应时间提升为 1s 了，那么一个营业员可能每分钟可以处理 3 笔业务，那么系统在业务最繁忙时段的忙时集中系数会大幅增

加，对系统资源的需求会大幅增大，这时很可能导致某个资源比较紧张的系统出现过载，甚至出现系统崩溃。

以前处理过一个优化项目，当时也是遇到了资源十分紧张的情况，系统十分缓慢，我建议客户每个营业厅关闭一个窗口，虽然每个窗口排的队更长了，但关闭了一些终端，系统资源的消耗下降，每个营业窗口处理业务的速度大幅提升，平均每个客户的等待时间反而下降了。

优化小技巧 DB Cache 调整方案如何制订

今天做优化方案时谈到了通过加大 DB Cache 来提升系统性能。这个优化方案看上去好像很简单，但是为什么要加大 DB Cache？加大以后会带来多大的好处？这是怎么评估出来的？大家可能很容易通过命中率来判断 DB Cache 是否足够，我也经常看 DB Cache 的命中率。而从调优的角度来看，命中率是十分粗略的东西，只能作为参考。

很多文档都说 DB Cache 的命中率要超过 95%，否则需要加大 DB Cache。Oracle 公司的多个健康检查的脚本把命中率定为 90%。到底 DB Cache 的命中率多少合适呢？这个问题恐怕无解。不同的系统在不同的情况下对命中率的要求是不同的。除了考虑命中率指标外，DB Cache 设置是否合理主要取决于当前的 DB Cache 大小是否产生了过多的物理 I/O,导致了物理 I/O 负载过大，以及是否加大了相关的闩锁竞争，导致了 DB Cache 相关的性能问题。而且，如果使用了多缓冲，简单的命中率并不能说明问题，不同缓冲区的命中率都是需要分别计算的，否则无法得到某个缓冲区的实际命中率。对于 keep 池，我们的命中率目标是 100%，如果低于 100%，就说明 keep 池过小。在很多情况下，keep 池的命中率低于 99%就可能存在较为严重的问题了。

更高级一点的分析是通过 OWI（Oracle Wait Interface）去分析，看看哪些等待事件是最为严重的，是不是存在较为严重的 I/O 等待，如何能够减少 I/O 等待，从而提高系统性能。如果系统中的 db file sequence read 等待十分严重，那么是不是能够加大 DB Cache 来减少这方面的等待，从而提高性能呢？哪怕命中率达到了 98%，加大 DB Cache 可能仍然能够取得较好的优化效果。通过 OWI 分析可以从等待事件、闩锁等情况分析 DB Cache 的设置是否存在问题。

如果你已经熟练掌握了 OWI 分析，那么下一步应该学习更高级的分析方法，就是一种从系统整体出发的分析方法。结合 OWI、时间模型、系统综合分析，从系统优化的角度来分析 DB Cache 是否需要调整。做数据库优化，实际上是对系统的 CPU、内存、I/O 资源进行优化组合。这三者之间的关系如下所示。

- 加大 DB Cache（内存），可以减少 I/O，但是会增加 CPU 的消耗。
- 减少 DB Cache，可以增加 I/O，减少 CPU 的消耗。

一个优秀的 DBA 应该能够在调整时，充分考虑系统资源的限制，在使任何一种资源都不出现短板的情况下提高系统的总吞吐能力。如果当前系统的 I/O 响应时间很正常，I/O 性能很好，那么加大 DB Cache 对系统整体性能的提升是微乎其微的。而对于一个 I/O 相对出现了瓶颈的系统，加大 DB Cache 往往可以获得较好的性能改善。而对于一个 CPU 已经十分紧张的系统，盲目加大 DB Cache，可能会导致严重的性能问题。一名优秀的 DBA 应该对这些有一个综合性的评估。

作为 DBA，应该十分清楚自己所做的优化目的是什么。是提高整个系统的吞吐能力，还是提高某个业务模块的响应速度？在系统资源很充分的时候，提高某个模块的响应速度是没有多大难度的，但是如果系统资源很紧张，片面去提高某个模块的性能可能会带来很恶劣的结果。

很多 DBA 总是通过一些所谓的理论来支持自己的工作。实际上做优化和练武术一样，练习招术是很初级的阶段，到了某个阶段，无招胜有招。如果达到了这种境界，对系统进行任何操作都能确保结果是好的。

举例来说，我的一个客户有个业务，即对一张超过 1GB 的表频繁进行随机访问，而且这个随机访问要求的响应时间十分严格。SQL 是最优的了，但是响应速度仍然达不到要求。考虑到当时 CPU 使用率还较低（不到 50%），内存也有大量的空闲，我建议将这张表和相关索引放入 keep 池。这样操作后，系统响应时间达到了客户的要求，CPU 的使用率上升了十几个百分点。按常理，这个操作是不“正确”的，也是不建议的，但是在这种场合，这个操作是唯一的解决方案。

以我这些年做性能优化的经验，操作层面上的东西是很简单的，但是要达到无招胜有招的境界，你必须十分清楚 Oracle 的基本原理，你要明白每个操作可能产生的后果，并且从中排除主要风险。有一次我们给客户做优化，是我们的一个资深工程师实施的，通过优化，系统响应时间提高了 10 倍，CPU 使用率从 100%下降为 50%左右。后来 Oracle 公司的一个朋友问我，怎么做的，我说很简单，做表分析，要求客户把 SQL 中的提示全部去掉就好了。他后来很感慨，因为这个项目 Oracle 公司也介入过，但是没有很好的解决方案。他说，做这些操作很容易，但是敢于做这些操作并且能够肯定地知道做了这些操作能解决问题的人才是真正的高手。

确实是这样，真正的高手并不是拥有一些所谓的独门秘笈的人，而是能够做出合理判断的人。

要说明的是，DB Cache 调整是一个十分复杂的问题。今天给大家介绍的是 DB Cache 优化的一些原则，并且指出了 DB Cache 优化时的一些误区。至于如何分析 DB Cache 并做出合理的设置，将在今后的“优化小技巧”中详细介绍，请大家不要着急。

优化小技巧　自动段存储空间管理 3

通过前面的分析，我们可以得出如下结论。

- 如果 LMT 设置为 AUTOALLOCATE 属性，那么段的默认扩展参数为 64KB，当段扩展总空间超过 1MB 后，按照 1MB 进行扩展（推断：如果段的空间超过 128MB，则按照 128MB 扩展）。
- Oracle 数据库通过 L3 BMB 来实现位图管理。L3 是最高级的 BMB，其次是 L2，L1 是最底层的 BMB。只有 L1 BMB 包含块的位图，其他级别的 BMB 只保存下级 BMB 的地址。
- L1 BMB 是成组出现的，至少包含一个块，每个 L1 BMB 最多包含 64 个块的位图。如果一个扩展超过 64 块，那么每组 L1 BMB 由 2 个或以上的 L1 BMB 组成。每个 L1 BMB 都包含一个指向所属的 L2 BMB 的指针。
- L1 BMB 后面是第一组 L2 BMB，每组 L2 BMB 可以包含 1 个或多个 L2 BMB（一般情况下，一个 L2 BMB 已经能够存储所有的 L1 BMB 信息）。

- L2 BMB 后面是第一组 L3 BMB（在本例子中，由于数据量不足，所以没有 L3 BMB）。
- L3 BMB（或者 L2 BMB）后面是段头信息（PAGETABLE SEGMENT HEADER），段头信息中包含第一个 L3 BMB 的指针、高水位信息（Highwater）、段扩展列表和 L2 BMB 的地址列表。
- 位图信息中的取值包括下面几个。
  - METADATA：BMB 或者段头块。
  - UNFORMATTED：未使用的块。
  - FULL：满块。
  - FS4：填充率 75%以上的块。
  - FS3：填充率 50%～75%的块。
  - FS2：填充率 25%～50%的块。
  - FS1：填充率 25%以下的块。

5月27日 无　　奈

今天是三方例会的日子，由于要讨论 6 月 6 日实施的一些细节，所以小齐也特意从北京赶过来。会议的地点也安排在了浑南的公司总部。这种安排说明今天会议的规格很高，孙主任可能会亲自参加今天的例会。

一进会议室发现开发商的项目总监范总已经先到了，他也是昨晚才赶到沈阳的。我们进去时正坐在孙主任旁边，和孙主任谈得挺投机。今天的会议由孙主任亲自主持，他首先阐述了 6 月 6 日进行第一次实施的必要性以及公司领导对这次优化实施的期望，希望我们几方能够通力协作，把这次优化实施成功地完成。

孙主任讲完后，小齐代表乙方谈了本次优化实施的情况。小齐讲话的前半段强调了本次优化实施并不是我们计划中的，存在较大的风险。后半段她也代表优化实施方表态，我们一定会精心准备，认真实施，尽我们最大的可能创造一个奇迹。小齐讲话结束后，老于介绍了我们针对 6 月 6 日的实施方案。在实施方案介绍完毕后，老于对开发商提出了需要开发厂商配合的几个方面。

- 在 6 日前上线部分已经完成优化的 SQL，以减轻 CPU 的负担。
- 5 日晚上完成一次数据库全备份。
- 根据我们提交的从未使用的索引清单，找出确认未使用的索引，并删除这些索引。
- 提取出一些关键业务模块，并在 6 日前将关键义务模块的主要性能指标以及执行计划都采集出来，6 日优化实施完毕后，对这些模块进行测试，并比对性能指标，以便找到性能有所下降的模块进行整改。
- 优化完成后，对主要业务功能模块进行功能测试。

本来不准备做索引删除操作的，不过我们估计开发商肯定会讨价还价，所以还是把这一条放在里面，作为讨价还价时做出让步的条款。除了这条外，其他条款是开发商必须做的，如果开发商拒绝其中的某个条款，6 月 6 日实施的风险就会加大。

范总看样子准备得十分充分。老于刚刚介绍完我们的方案，他立即做出了回应。

- 由于开发流程所限，SQL 优化最早能够在 6 月 20 日之前提交成果，6 日前没有一个应用可以上线。
- 为了保证优化的实施，5 日可以配合做一个数据库全备份，并且对应用服务器和数据库的操作系统也同时做一个全备份。
- 索引问题十分复杂，目前开发部门正在审核。要剔除一个索引的难度很大，必须经过严

格的测试，由于这一点是我们今天才提出来的，所以他还无法做出安排，具体时间明天可以给我们答复。

- 关于关键业务模块，因为本系统中任何模块都很关键，如果找出来就很多了，并且现场的技术人员能力不足，无法对其性能指标做出采样，所以还是希望我们协助采集这些数据。
- 关于功能测试，由于模块有几百个，人力资源不足，时间上来说也来不及，所以只能做一些简单的测试。
- 另外要强调的是，由于准备不足，开发商无法对优化结果承担任何责任，希望优化小组尽可能完善优化方案，以防备出现性能倒退。开发团队在 7 日会安排现场项目经理小王 8：00 前赶到现场，协助优化小组处理可能出现的问题。

这应该是最差的结果了，我们精心准备的一拳仿佛被一个武林高手用太极神功化解了。开发厂商拒绝了所有要求，并且把他们和这个优化项目割裂开来。从目前来看，在今天的会议上我们几乎没有得到任何实质性的东西。看样子这次优化，他们扮演的只是一个看客的角色。如果我们做好了，他们也许会上来拍拍肩膀说："兄弟，你真行，干的不错。"如果我们干砸了，那就很难说了。范总的讲话结束后，大家沉默了几分钟。

再谈下去也没多大意义了，会议刚刚开始不到半个小时，孙主任就宣布散会，同时让我们留下来，和张工继续讨论实施细节。看样子，孙主任的情绪也不高。我们建议本次实施分三天完成，每个晚上只完成一套系统。停机窗口从晚上 8：00 到第二天早晨 7：00，如果 5：00 还无法完成，就必须回退。当天晚上的统计作业延后到第二天中午 12：00 开始执行，可能会影响第二天下午的系统性能，数据库归档日志的备份当天晚上以及第二天下午暂停。第二天早上我们和客户各安排一个工程师在现场值班，其他人上午休息，下午晚点过来，以便留好体力干第二天晚上的活。

我们正和张工讨论，孙主任送走了开发厂商的人，又回到会议室说："我们临时改变计划，给你们添麻烦了。由于准备比较仓促，这次调整肯定有风险，做好了大家都好，出了问题算我的，大家别有太多的心理负担。"说完向大家摆摆手："我还有个会，先走一步。"刚走到门口，孙主任又回过头来说："不过哥几个，准备尽可能充分一些，别让我太为难。"

走出院子，看到有一辆出租车正在下客，我连忙让老熊和老肖先坐车回去，我和小齐、老于再等下一辆。等了 5 分钟，还是没见空车过来，这个时候张工的电话打过来了，小齐接了电话后十分感慨："刚才孙主任把范总叫到办公室谈了半天，希望他们能配合加快应用优化的进度，配合我们 6 日的优化操作。两个人谈的很诚恳，不仅仅是以工作关系谈的，孙主任是以朋友的身份请范总帮忙。不过范总那边确实有难处，原本的优化计划是 20 日之前上线，这回要提前半个月，他们的人力资源和技术能力都有限，所以他们不敢承诺。为此，孙主任让张工转告我们，他今天说的话都是发自肺腑的，大家放心干，只要尽力了，出了问题也不怕。能碰到这种客户，算是我们走运吧。"

确实也是，这些年来我见过的客户起码也上百了，能够这么说这么做的客户确实不多。为了这样的客户，大家伙多流点汗也值啊。虽说孙主任的话让我们很感动，不过感动过后，大家感到

了更大的压力。6 日优化后如果效果不大，会让用户对我们的信心有所下降，也会给应用开发厂商更多打太极拳的借口。反过来，如果这次优化的效果很好，那也不是什么好事。因为一下子优化效果很明显，从表面上看，系统的问题解决了，那样也可能给开发商拒绝修改 SQL 带来借口。所以说这次优化不仅要避免优化效果不足导致系统性能下降的问题，也要避免优化效果过好导致开发商修改 SQL 的动力下降的问题。

中午吃饭时，针对优化效果过好的问题我们展开了讨论。最后的结论是，重建索引的工作我们暂时不做。如果 7 日发现系统性能有所提升，可以达到我们的目的，那么这个工作押后到第二次优化调整时做；如果 7 日发现系统性能还存在一定的问题，我们的优化效果没有达到，那么 7 日白天和晚上我们可以陆续手工进行索引重建。由于 7 日是周末，系统负载比平时小，只要我们把索引按照大小进行分类，白天做小索引的重建，晚上做大索引的重建，应该问题不大。

今日点评

今天老白他们希望开发商配合的事情全部被开发商否决了，这种情况其实在优化项目中经常碰到。实际上在这个项目刚刚开始时，老白就已经感觉到开发商可能配合不力，因此早就做好了脱离开发商独立工作的准备。优化项目往往都是上线时间比较长的系统，处于维护期的中后段，可能再坚持一两年就会退网或者升级了。对于开发商来说，这种系统往往只留下很小的一支队伍在对系统进行维护，这支维护队伍往往只解决必须解决的一些 bug 和开发一些新模块以满足客户需求。在这种系统中，很多模块的开发人员可能都已经离开了现场，甚至离开了公司，所以对这些模块的修改是开发人员最不愿意面对的。因此 DBA 对于开发人员也要给予足够的理解，也要有充分的准备，在开发人员不愿意修改程序的情况下，拿出应急方案来。

对于一般的项目，如果没有开发商的配合，也可以通过系统级的调整，并通过表分区、索引优化、表碎片整理等方式达到优化的目的。这种情况也是老白经常遇到的。在无法得到开发商配合的情况下独立完成优化也是 DBA 经常要面对的。而对于沈阳这个项目，想要完全脱离开发商的支持，效果肯定不会很好。因此，老白他们面临在第一次优化实施时开发商完全不配合的局面时，需要更为细致地准备优化方案。既要在开发商不能配合的情况下，尽可能地提升系统性能，达到优化目的，又要控制好优化的效果。所谓控制好优化效果，就是第一次优化不能用光所有的手段，对效果一定要有所保留，以防第一次优化效果过好，导致开发商更不愿意修改应用。这种优化项目，开发商修改应用是长期保持优化效果的关键，因此必须逼着开发商修改应用，否则这个项目就会前功尽弃。

优化小技巧　临时表空间和排序

临时表空间和排序操作是数据库中最常见的操作，也是绝大多数 DBA 最为困惑的操作之一。本节内容就是要向 DBA 介绍临时表空间以及在临时表空间中发生的主要操作。本节并不是讨论临时段的章节，因此需要了解临时段的读者，可以参考本书下一节的相关内容。本节讨论的临时

段是指在临时表空间中，排序等操作使用的临时空间。

对于临时表空间的管理机制和临时段的算法，Oracle 7.3 是一个革命性的版本。由于 Oracle 7.3 推出的 OPS 机制，原有的临时段算法无法适应 OPS 的需要，所以 Oracle 公司推出了一个全新的临时段管理算法。在 Oracle7.3 之前，临时段是在需要时分配，使用完毕后删除的。Oracle 7.3 推出了一种全新的临时表空间的管理算法。新算法的核心就是 SEP（SORT EXTENT POOL），它用来管理临时段中扩展的结构，存储在共享池内。任何需要使用排序空间的操作都需要从 SEP 中分配空闲的扩展。使用完毕后，该空间不需要释放，只要在 SEP 中将该扩展设置为空闲即可。

当数据库实例启动后，SMON 将会进行一个操作，删除该实例的临时段，并且对临时表空间进行碎片整理。在这个操作完成前，数据库打开的操作不能完成。因此每次数据库重启后，临时段中的垃圾会被完全清理。当数据库打开后，第一个进行的硬盘排序的操作会在相关的临时表空间内创建临时段，这个临时段也是整个实例唯一的临时段（在新的临时段算法下，同一个表空间内，每个实例只有一个临时段）。临时段中扩展的信息会被记录在 SEP 中。硬盘排序操作会在 SEP 中查找可用的扩展，在查找前，需要获得 `SORT EXTENT POOL` 闩锁。如果能找到可用的扩展，那么 SEP 中被分配的扩展将会被标注为占用状态；如果找不到可用的扩展，那么就会试图从表空间中分配新的空间，如果这个分配工作因为表空间的空闲空间不足而无法完成，那么会产生一个 ORA-1652 错误。

当排序操作完成时，会再次获取 `SORT EXTENT POOL` 闩锁，并且将使用的扩展标注为空闲，然后释放 `SORT EXTENT POOL` 闩锁。

新的临时表空间管理算法不需要频繁地分配和释放临时段，大大提高了临时段管理的效率。由于这种只分配不释放的算法，所以 DBA 经常会看到自己的临时表空间总是处于或者接近 100% 使用的状态。其实对于 Oracle7.3 以后的版本，临时表空间使用率接近 100%是十分正常的。DBA 可以通过 `v$sort_usage` 和 `v$sort_segments` 这两个视图来检查临时段的使用情况。

在 RAC/OPS 环境下，临时段的管理算法也是类似的。由于在 RAC 中多个实例会共享一个临时表空间，所以多个实例也能够共享相同的扩展。在 RAC 环境下，每个实例拥有独立的 SEP，每个实例中的排序操作需要在自己的 SEP 中分配空间，因此如果 SEP 中无法分配到足够的空间，就会先在表空间中分配空间。如果表空间的空间已经分配完毕，且别的实例还有空闲的扩展，那么这个扩展可以分配给需要的实例使用，这些操作是不可见的，服务器进程不会收到 ORA-1652 的错误信息，但是在 `ALTER LOG` 中会有一个 ORA-1652 的记录。

优化小技巧　undo 表空间和回滚段

`undo/rollback` 表空间是用于存放回滚段（rollback segment）的表空间。回滚段是 Oracle 用于保存被修改数据的前映象的数据空间。每个回滚段包含一些扩展，而回滚段采用一种循环机制来使用这些扩展。当某个扩展写满后，自动切换到另外一个扩展继续使用。一个事务会将回滚记录写在回滚段的当前位置，并且通过记录大小来标明记录的位置。当前写指针是回滚段段头中的一个控制结构。尾部指的是回滚段中最后一条记录的开始位置。

回滚段的数量和每个回滚段的大小对于回滚段的配置来说至关重要。在 OLTP 系统中，设置足够的回滚段数量，以避免回滚段冲突。每个回滚段的大小应该足够大，以便于适应事务处理的需要。Oracle 循环使用回滚段，所有的在线回滚段（除了系统回滚段外）都会被轮流使用。Oracle 在使用回滚段时，有以下特点。

- 一个事务只使用一个回滚段来记录所有的回滚记录。
- 多个事务可以写入相同的扩展。
- 扩展被循环使用，任何一个扩展都不会被跳过。
- 如果不能使用下一个扩展，Oracle 会自动分配一个新扩展，并将其插入这个环中。
- Oracle 不会使用被尾部占据的扩展。

从上面的原则可以看出，事务的持续时间和事务的大小同样是影响回滚段的重要因素。一个哪怕只使用了一个字节的长时间的事务也可能导致回滚段扩展。如果一个系统中有多个回滚段，那么 Oracle 会根据下面的原则来选择一个回滚段。

- 首先，Oracle 会选择当前事务数最少的回滚段。
- 如果有两个以上的回滚段内当前有相同的事务数，那么 Oracle 会根据上一个事务使用的回滚段号，选择比这个回滚段号大的下一个顺位的回滚段。比如当前事务使用了回滚段 2，回滚段 1 和回滚段 4 有最少的事务数，那么 Oracle 会使用回滚段 4 来存储下一个事务的数据。

从上面的原则，我们能够在设置自己的回滚段时得到什么启发呢？如果你的回滚段足够大，那么首先需要根据实际的事务大小来确定回滚段的存储参数，使回滚段头不会很快转到段尾。如果回滚段很快转到尾部，就会导致回滚段很快进行扩展。第二个应该注意的问题是，如果执行一个长时间运行的查询，这个查询访问的数据变化频率十分快，那么就需要保证回滚段的大小足够大，以保证你的查询结束之前不会出现 ORA-1555 错误（Snapshot too old ）。

回滚段的大小取决于数据库中事务的特点。回滚段大小的设计应该满足数据库日常事务的要求，而不是针对那些不经常使用的大的事务。回滚段的数量设置要保证不会发生回滚段竞争。是否有回滚段竞争可以通过 `v$waitstat` 视图来查看，代码如下：

```
SELECT CLASS, COUNT FROM v$waitstat WHERE CLASS = '%undo%';
```

其中任何非 0 的 `COUNT` 值的存在都说明存在回滚段头的竞争。

5月28日 BBED 的妙用

今天刚到办公室，就看见开发商的项目经理小王焦急地在我们的办公室里打转。一看到我走进来了，马上满脸笑容地迎了上来："白哥，你可来了，我这边有个奇怪的问题请您帮个忙。"

"如果和优化项目有关的，我肯定鼎力相助，无关的事情我可帮不上忙"，我一副公事公办的样子。

"白哥，你这就不够哥们了，这个项目配上我应该算够意思吧，今天你帮我搞定这个问题，今后需要我们配合的地方我能不向着你吗？"

我看着小王急得抓耳挠腮的，知道也逗得差不多了，就说："兄弟，这可是你说的，千万别食言啊。"

原来小王他们负责的一个项目，今天要完成项目的交付前测试，明天提交测试版本。但是在他们测试环境上要创建几张新表时总是报错，使他们今天的测试无法正常进行。我检查了一下系统，发现系统的工作基本正常，数据库的读写都没什么问题，不过当他们用 imp 导入测试数据时，突然开始报 ORA-8102，无法导入数据。

我和小王到了他们的办公室，让他们演示了一下导入过程，导入第一张表时没有报错，而导入第二张表时就出现了 ORA-8102，检查 Trace 文件，发现：

```
*** 2006-05-28 08:59:02.369
*** ACTION NAME:() 2006-05-28 08:59:02.369
*** MODULE NAME:(SQL*Plus) 2006-05-28 08:59:02.369
*** SERVICE NAME:(SYS$USERS) 2006-05-28 08:59:02.369
*** SESSION ID:(150.11795) 2006-05-28 08:59:02.369
oer 8102.2 - obj# 49, rdba: 0x0200557a(afn 8, blk# 21882)
kdk key 8102.2:
  ncol: 1, len: 5
  key: (5):  04 c3 45 41 64
```

我发现 ORA-8102 是在创建表时出现，为了找到 imp 操作中出错的部分，我们需要取出刚才报错的那张表的建表脚本。可以通过下面的方法获得该建表脚本：

```
imp show=y  full=y >impdata.log
```

这样 imp 操作就把所有的建表脚本都写入 impdata.log 了。打开 impdata.log，取出第二个表的建表语句，直接在 SQL*Plus 上执行，报相同的错误，看样子确实是在建表时出错了。从日志上看，出错的对象的对象号（obj#）是 49。我们需要知道 49 号对象是什么，于是执行下面的语句：

```
SQL> SELECT name,type# FROM obj$ WHERE obj#=49;
NAME                                TYPE#
------------------------------ ----------
I_CON2                                  1
```

其中 I_CON2 是一个索引，我们检查一下这个索引是属于哪张表的具体代码如下：

```
SQL> SELECT table_name FROM dba_indexes WHERE index_name='I_CON2';
TABLE_NAME
------------------------------
con$
```

其中 con$表是一个系统表，存放约束关系数据。在 SQL*Plus 上通过 10046 的 Trace 文件来确认一下：

```
SQL>ALTER SESSION SET EVENTS='10046 immediate trace context forever,level 10';
```

然后再执行那个建表脚本，在产生的日志文件里，我们发现如下信息：

```
=====================
PARSING IN CURSOR #3 len=51 dep=1 uid=0 oct=6 lid=0 tim=839547263840 hv=2404924642
ad='cef1d768'
update con$ set con#=:3 where owner#=:1 and name=:2
END OF STMT
PARSE #3:c=0,e=386,p=0,cr=0,cu=0,mis=1,r=0,dep=1,og=4,tim=839547263834
BINDS #3:
kkscoacd
 Bind#0
  oacdty=02 mxl=22(22) mxlc=00 mal=00 scl=00 pre=00
  oacflg=08 fl2=0001 frm=00 csi=00 siz=24 off=0
  kxsbbbfp=800003ffbfdeac50  bln=24  avl=03  flg=05
  value=686500
 Bind#1
  oacdty=02 mxl=22(22) mxlc=00 mal=00 scl=00 pre=00
  oacflg=08 fl2=0001 frm=00 csi=00 siz=24 off=0
  kxsbbbfp=800003ffbfdeac80  bln=22  avl=01  flg=05
  value=0
 Bind#2
  oacdty=01 mxl=32(16) mxlc=00 mal=00 scl=00 pre=00
  oacflg=18 fl2=0001 frm=01 csi=871 siz=32 off=0
  kxsbbbfp=c0000000cef1ed2e  bln=32  avl=16  flg=09
  value="_NEXT_CONSTRAINT"
oer 8102.2 - obj# 49, rdba: 0x0200557a(afn 8, blk# 21882)
kdk key 8102.2:
  ncol: 1, len: 5
  key: (5):  04 c3 45 41 64
```

从日志上看，是在做修改 con$表时出的错。从后面的绑定变量列表里看，有 3 个绑定变量。第一个绑定变量是给 con#赋值的，从 Bind#0 来看，这个值是 686500。第二个绑定变量是 owner#的 WHERE 条件，值是 0，也就是对应系统账号。第三个绑定变量是 name 的取值，我们可以看到正好是_NEXT_CONSTRAINT。这个 SQL 的含义是将_NEXT_CONSTRAINT 对应的 con#值设置为 686500。这个语句是 con$记录生成算法中的一部分。根据生成约束关系记录的算法，首先从 con$表中找到 NAME=_NEXT_CONSTRAINT 的记录，作为新产生的 CONSTRAINT

的主键 con#的取值，再修改 con$表，将_NEXT_CONSTRAINT 的 con#加 1。

通过跟踪我们可以看到，UPDATE 的 con#参数是 686500，那么修改前这个值是 686499，翻译成 Oracle 内部存储格式就是 c3 45 41 64（数值类型的值转换为 Oracle 内部存储格式的算法请参考今天的“优化小技巧”）。这就是从 Trace 中看到的出现 oer 8102 的那个 key。根据 Trace 信息可以看到 rdba:0x0200557(8/21882)是一个叶节点块：

```
Block header dump:  0x0200557a
 Object id on Block? Y
 seg/obj: 0x31  csc: 0x00.23049f4  itc: 5  flg: 0  typ: 2 - INDEX
     fsl: 0  fnx: 0x200557b ver: 0x01
```

该块属于 0x31 对象，也就是 obj#=49 的对象（I_CON2）。TYP:2-INDEX 说明这个块是索引块。从块头来看，块并无异常。该块拥有 193 条记录（在索引块的转储文件中，kdxconro 对应的值就是本块中的记录数，本案例是 193），每条记录对应一条 con$表的记录，如下：

```
col 0; len 4; (4):  c3 45 3f 21
row#185[5374] flag: ---D--, lock: 3, len=13, data:(6):  02 00 69 21 00 44
col 0; len 4; (4):  c3 45 3f 22
row#186[5361] flag: ---D--, lock: 3, len=13, data:(6):  02 00 69 21 00 43
col 0; len 4; (4):  c3 45 3f 23
row#187[5348] flag: ---D--, lock: 3, len=13, data:(6):  02 00 69 21 00 41
col 0; len 4; (4):  c3 45 3f 24
row#188[5335] flag: ------, lock: 0, len=13, data:(6):  02 00 69 21 00 23
col 0; len 4; (4):  c3 45 41 5b
row#189[5322] flag: ------, lock: 0, len=13, data:(6):  02 00 69 21 00 1f
col 0; len 4; (4):  c3 45 41 5c
row#190[5309] flag: ------, lock: 0, len=13, data:(6):  02 00 69 21 00 1e
col 0; len 4; (4):  c3 45 41 5d
row#191[5296] flag: ------, lock: 0, len=13, data:(6):  02 00 69 21 00 22
col 0; len 4; (4):  c3 45 41 5e
row#192[5166] flag: ------, lock: 0, len=13, data:(6):  00 40 00 aa 00 11
col 0; len 4; (4):  c3 45 42 0e
```

按常规，_NEXT_CONSTRAINT 应该在 con$表刚刚建立时就已经产生，因此该记录应该位于 1 号文件。从这 193 条记录来看，只有最后一条记录 0x004000aa00-11（1/43520 第 11 号记录）位于 FILE 1。由于_NEXT_CONSTRAINT 的 con#是下一条 con$记录的 con#的值，所以该记录应该是该表中的 con#最大的记录，理应处于索引的最后一行。而从该行的值来看 c3 45 42 0e，翻译成十进制值应该是 686514，这个值明显和 686499 不同。这就是导致 ORA-8102 的主要原因。

看样子问题就出在索引上的键值和表上的键值不同上了。由于这张表上出现的这个故障，创建带约束关系的表就会出错，而创建普通表就没有问题，所以第一张不带约束关系的表能够创建成功，而第二张带约束关系的表就无法成功创建了。我想如果只是索引损坏那还好办，直接重建一下就行了，如下：

```
SQL> ALTER index i_con2 rebuild;
ALTER index i_con2 rebuild
*
```

```
ERROR at line 1:
ORA-00701: object necessary for warmstarting database cannot be altered
```

看到报错我才想起来，i_con2 是一个 bootstrap$对象（关于 bootstrap$对象，请参考今天的今日点评），而且 obj#小于 51，不仅仅是 bootstrap$对象，而且是核心 bootstrap$对象，连 38003 事件都不会起作用。

我告诉小王，由于 con$表上的索引故障，所以无法创建新的带约束关系的表了。建议他们把数据库重建一下，再重新导入一下数据就没问题了。目前只是无法创建新的约束关系，其他操作不会受到影响。

小王很着急地问，还有没有其他办法。实际上他们今天只需要导入 10 多张表的数据，其他数据是已经准备好的。如果重建数据库，那么前一阵子他们准备好的 800 GB 数据就需要重新导出再导入进去。在这台 P650 上，做这个事情起码要折腾一整天。而且要导出 800GB 的数据，现在测试机上的磁盘空间都不够。

这就难办了，如果不重建数据库，索引又不能重建，还能有什么办法呢？看样子唯一的办法就是使用 BBED 了。通过 BBED 将索引和表的键值修改为一致的就可以解决问题了。BBED 是 Oracle 的数据块修复工具，可以用来对数据块进行分析和修改。由于该工具直接修改数据块的内容，所以在生产环境中是不敢轻易使用的。我做 DBA 这么多年以来，用 BBED 解决问题也只有寥寥可数的几次。今天这个库是个测试库，风险不大，拿来练练手还是不错的。

经过考虑，我决定通过修改索引上的数据，使索引和表一致。这条记录的位置是在数据块中偏移量 5166 开始的位置（5166 是一个相对偏移量。实际上通过 5166 这个值，可以计算出这条记录在块中的绝对位置。计算绝对位置的方法参考今日的“优化小技巧”）。在这一行数据中，第一个字段是 ROWID，共 6 个字节，从第 7 个字节开始就是这条索引的键值。从 Trace 中我们看到的键值是 c3 45 42 0e，我们要把键值修改为 c3 45 41 64，也就是说要把 5174-5175 字节的两个值从 42 0E 修改为 41 64。确认计算的偏移量没有错误后，我使用 BBED 修改了数据，然后重启数据库，再执行建表脚本，问题消失了。

小王看着我眼花缭乱的操作，在旁边眼都直了。他说：“白哥，学你这个难不？要不你教我算了，有这个手艺，今后混口饭吃就不愁了。”

我说：“你小子还是先帮我们尽快把几个 SQL 修改好吧。如果你小子不卖力气，别怪我不客气。”

今日点评

今天老白帮助小王他们解决了一个问题。在做优化项目时，帮助开发团队解决一些和项目无关的问题，对 DBA 来说一般是稍带手的事情。而这对于改善优化团队和开发团队的关系很有用，如果碰到这种机会一般来说一定要抓住。今天小王欠了老白一个人情，在后面的优化实施过程中，小王和老白的配合就更容易了。

今天老白通过了一个非常规的手段，通过 BBED 修改数据解决了问题。使用 BBED 修改生产库是一件十分谨慎的事情，一定要有足够的把握才可以实施。由于今天是在一个测试环境中，所以老白今天就没有采取备份等手段。如果在生产环境中，一定要首先备份要修改的数据文件才能实施操作。

今天碰到的新概念还是很多的，比如说 bootstrap$对象的概念。大家可能对 bootstrap$对象十分陌生，其实搞过软件开发的人都知道，一个系统在刚刚开始时要对系统进行初始化，包括初始化缓冲区、变量、常量等，这个过程就是 bootstrap 过程。Oracle 也有一个 bootstrap 过程，就是对 Oracle 数据库进行初始化。和初始化过程相关的对象我们称为 bootstrap 对象。这些对象的创建脚本都存放在 bootstrap$表中。这些对象是不能随便重建的，哪怕是索引也不能随便重建。bootstrap 对象分为核心 bootstrap 对象和非核心 bootstrap 对象。如果非核心 bootstrap 对象出了问题，可以通过设置 EVENT 38003（关于 EVENT 38003 的详细情况，请参考 Metalink Note 279797.1），使系统启动时不把这些对象作为 bootstrap$对象处理。而这个事件对于核心 bootstrap$对象是不起作用的。

优化小技巧　如何启用 BBED

BBED 是 Oracle 修改数据块的一个常用工具，从 Oracle 7.3.2 就可以提供了，用来对那些损坏的数据库进行修复。BBED 本身已经随 Oracle 安装介质安装了，只是在一般环境下没有启用。在 Windows 下，BBED 一般在$ORACLE_HOME/bin 目录下。在 Unix 下，BBED 默认情况下并没有安装，不过在 Unix 平台下，可以使用以下命令产生：

```
cd $ORACLE_HOME/rdbms/lib
make -f ins_rdbms.mk $ORACLE_HOME/rdbms/lib/bbed
```

或者

```
cd $ORACLE_HOME/bin
make -f ins_rdbms.mk BBED=$ORACLE_HOME/bin/bbed $ORACLE_HOME/bin/bbed
```

BBED 命令可以使用 HELP=Y 来查看帮助，具体如下：

```
BBED HELP=Y
PASSWORD - Required parameter
FILENAME - Database file name
BLOCKSIZE - Database block size
LISTFILE - List file name
MODE - [browse/edit]
SPOOL - Spool to logfile [no/yes]
CMDFILE - BBED command file name
LOGFILE - BBED log file name
PARFILE - Parameter file name
BIFILE - BBED before-image file name
REVERT - Rollback changes from BIFILE [no/yes]
SILENT - Hide banner [no/yes]
HELP - Show all valid parameters [no/yes]
```

PASSWORD 参数是 BBED 工具的保护参数。启动 BBED 工具是需要口令的，这是 Oracle 公司

的一个保护措施，不希望 BBED 被非授权使用。不过 Oracle 公司针对 BBED 的口令已经是一个公开的密码了，在所有的 BBED 版本中，口令都是 blockedit。

FILENAME 是 BBED 需要处理的 Oracle 文件名。如果处理单个文件，可以使用这个参数；如果要处理多个文件，建议使用 LISTFILE 来代替这个参数。LISTFILE 对应的是一个文件名，比如 LISTFILE=mydbfile.txt。mydbfile.txt 的内容如下：

```
1 D:\ORACLE\ORADATA\ORA92\SYSTEM01.DBF
2 D:\ORACLE\ORADATA\ORA92\UNDOTBS01.DBF
3 D:\ORACLE\ORADATA\ORA92\DRSYS01.DBF
4 D:\ORACLE\ORADATA\ORA92\EXAMPLE01.DBF
5 D:\ORACLE\ORADATA\ORA92\INDX01.DBF
6 D:\ORACLE\ORADATA\ORA92\ODM01.DBF
7 D:\ORACLE\ORADATA\ORA92\TOOLS01.DBF
8 D:\ORACLE\ORADATA\ORA92\USERS01.DBF
9 D:\ORACLE\ORADATA\ORA92\XDB01.DBF
```

该文件名的格式是文件 ID 加上文件名。

MODE 有两个取值：一个是 browse，也就是仅仅浏览数据，不做修改；另外一个取值是 edit，就是要修改数据。如果你用 BBED 的目的是修改数据，那么一定要选择 MODE= edit。

BLOCKSIZE 是数据文件的块大小。

SPOOL=yes|no 的含义是是否要把操作过程写入日志。BBED 操作是十分重要的操作，因此建议大家使用 SPOOL=yes。

CMDFILE=<某个文件名>，启动 BBED 后直接执行的脚本，里面可以包含 BBED 命令。

LOGFILE=<某个文件名>，日志文件的名字。

PARFILE=<某个文件名>，参数文件的名字。如果你有很多参数，写成命令行不方便，可以编辑一个参数文件。

BIFILE=<某个文件名>，修改数据块前可以将修改前映像保存在一个文件里，以便于出问题后使用 BIFILE 的内容进行恢复。

REVERT=yes|no 表示是否可以使用 BIFILE 进行回退。

SILENT=yes|no 表示是否可以使用静默模式。静默模式适合于使用 CMDFILE 的批处理方式。

上面我们介绍了 BBED 的启动参数，下面我们通过一个小案例来学习如何使用 BBED。我们设计的场景是如果在系统数据文件中发现一个数据块的事务状态一直没有提交，则使用 BBED 来修改数据块中的事务状态。

```
BBED LISTFILE=listfile.txt BLOCKSIZE=16384

口令:
输入 blockedit
BBED: Release 2.0.0.0.0 - Limited Production on 星期二 4 月 15 02:04:35 2008

Copyright (c) 1982, 2002, Oracle Corporation.  All rights reserved.

************* !!! For Oracle Internal Use only !!! ***************
```

修改文件中的第 62 个数据块

```
BBED> set block 62;
        BLOCK#          62

BBED> p ktbbh
struct ktbbh, 72 bytes                      @20
   ub1 ktbbhtyp                             @20       0x01 (KDDBTDATA)
   union ktbbhsid, 4 bytes                  @24
      ub4 ktbbhsg1                          @24       0x00000012
      ub4 ktbbhod1                          @24       0x00000012
   struct ktbbhcsc, 8 bytes                 @28
      ub4 kscnbas                           @28       0xf72537a2
      ub2 kscnwrp                           @32       0x0009
   b2 ktbbhict                              @36       2
   ub1 ktbbhflg                             @38       0x02 (NONE)
   ub1 ktbbhfsl                             @39       0x00
   ub4 ktbbhfnx                             @40       0x00000000
   struct ktbbhitl[0], 24 bytes             @44
      struct ktbitxid, 8 bytes              @44
         ub2 kxidusn                        @44       0x0006
         ub2 kxidslt                        @46       0x0006
         ub4 kxidsqn                        @48       0x004af52e
      struct ktbituba, 8 bytes              @52
         ub4 kubadba                        @52       0x0080039c
         ub2 kubaseq                        @56       0xe891
         ub1 kubarec                        @58       0x01
      ub2 ktbitflg                          @60       0x2001 (KTBFUPB)
      union _ktbitun, 2 bytes               @62
         b2 _ktbitfsc                       @62       0
         ub2 _ktbitwrp                      @62       0x0000
      ub4 ktbitbas                          @64       0xf72537a5
   struct ktbbhitl[1], 24 bytes             @68
      struct ktbitxid, 8 bytes              @68
         ub2 kxidusn                        @68       0x0008
         ub2 kxidslt                        @70       0x0044
         ub4 kxidsqn                        @72       0x004a0e5e
      struct ktbituba, 8 bytes              @76
         ub4 kubadba                        @76       0x0080011d
         ub2 kubaseq                        @80       0xf935
         ub1 kubarec                        @82       0x01
      ub2 ktbitflg                          @84       0x8000 (KTBFCOM)
      union _ktbitun, 2 bytes               @86
         b2 _ktbitfsc                       @86       9
         ub2 _ktbitwrp                      @86       0x0009
      ub4 ktbitbas                          @88       0xf7251d0b
```

使用 map 命令查看当前数据块

```
BBED> map
 File: C:\oracle\ora92\datadb\SYSTEM01.DBF (0)
 Block: 62                                  Dba:0x00000000
------------------------------------------------------------
 KTB Data Block (Table/Cluster)
```

```
 struct kcbh, 20 bytes                     @0

 struct ktbbh, 72 bytes                    @20

 struct kdbh, 14 bytes                     @92

 struct kdbt[1], 4 bytes                   @106

 sb2 kdbr[210]                             @110

 ub1 freespace[841]                        @530

 ub1 rowdata[15009]                        @1371

 ub4 tailchk                               @16380


BBED> p kcbh
struct kcbh, 20 bytes                      @0
   ub1 type_kcbh                           @0        0x06
   ub1 frmt_kcbh                           @1        0x02
   ub1 spare1_kcbh                         @2        0x00
   ub1 spare2_kcbh                         @3        0x00
   ub4 rdba_kcbh                           @4        0x0040003e
   ub4 bas_kcbh                            @8        0xf72537a5
   ub2 wrp_kcbh                            @12       0x0009
   ub1 seq_kcbh                            @14       0x01
   ub1 flg_kcbh                            @15       0x06 (KCBHFDLC, KCBHFCKV)
   ub2 chkval_kcbh                         @16       0x3b24
   ub2 spare3_kcbh                         @18       0x0000

BBED> map /v
 File: C:\oracle\ora92\datadb\SYSTEM01.DBF (0)
 Block: 62                                   Dba:0x00000000
------------------------------------------------------------
 KTB Data Block (Table/Cluster)

 struct kcbh, 20 bytes                     @0
    ub1 type_kcbh                          @0
    ub1 frmt_kcbh                          @1
    ub1 spare1_kcbh                        @2
    ub1 spare2_kcbh                        @3
    ub4 rdba_kcbh                          @4
    ub4 bas_kcbh                           @8
    ub2 wrp_kcbh                           @12
    ub1 seq_kcbh                           @14
    ub1 flg_kcbh                           @15
    ub2 chkval_kcbh                        @16
    ub2 spare3_kcbh                        @18

 struct ktbbh, 72 bytes                    @20
    ub1 ktbbhtyp                           @20
    union ktbbhsid, 4 bytes                @24
```

```
      struct ktbbhcsc, 8 bytes              @28
      b2 ktbbhict                           @36
      ub1 ktbbhflg                          @38
      ub1 ktbbhfsl                          @39
      ub4 ktbbhfnx                          @40
      struct ktbbhitl[2], 48 bytes          @44

   struct kdbh, 14 bytes                    @92
      ub1 kdbhflag                          @92
      b1 kdbhntab                           @93
      b2 kdbhnrow                           @94
      sb2 kdbhfrre                          @96
      sb2 kdbhfsbo                          @98
      sb2 kdbhfseo                          @100
      b2 kdbhavsp                           @102
      b2 kdbhtosp                           @104

   struct kdbt[1], 4 bytes                  @106
      b2 kdbtoffs                           @106
      b2 kdbtnrow                           @108

   sb2 kdbr[210]                            @110

   ub1 freespace[841]                       @530

   ub1 rowdata[15009]                       @1371

   ub4 tailchk                              @16380


  BBED> map
   File: C:\oracle\ora92\datadb\SYSTEM01.DBF (0)
   Block: 62                                  Dba:0x00000000
  ------------------------------------------------------------
   KTB Data Block (Table/Cluster)

   struct kcbh, 20 bytes                    @0

   struct ktbbh, 72 bytes                   @20

   struct kdbh, 14 bytes                    @92

   struct kdbt[1], 4 bytes                  @106

   sb2 kdbr[210]                            @110

   ub1 freespace[841]                       @530

   ub1 rowdata[15009]                       @1371

   ub4 tailchk                              @16380


  BBED> p ktbbh
```

```
struct ktbbh, 72 bytes                    @20
   ub1 ktbbhtyp                           @20      0x01 (KDDBTDATA)
   union ktbbhsid, 4 bytes                @24
      ub4 ktbbhsg1                        @24      0x00000012
      ub4 ktbbhod1                        @24      0x00000012
   struct ktbbhcsc, 8 bytes               @28
      ub4 kscnbas                         @28      0xf72537a2
      ub2 kscnwrp                         @32      0x0009
   b2 ktbbhict                            @36      2
   ub1 ktbbhflg                           @38      0x02 (NONE)
   ub1 ktbbhfsl                           @39      0x00
   ub4 ktbbhfnx                           @40      0x00000000
   struct ktbbhitl[0], 24 bytes           @44
      struct ktbitxid, 8 bytes            @44
         ub2 kxidusn                      @44      0x0006
         ub2 kxidslt                      @46      0x0006
         ub4 kxidsqn                      @48      0x004af52e
      struct ktbituba, 8 bytes            @52
         ub4 kubadba                      @52      0x0080039c
         ub2 kubaseq                      @56      0xe891
         ub1 kubarec                      @58      0x01
      ub2 ktbitflg                        @60      0x2001 (KTBFUPB)
      union _ktbitun, 2 bytes             @62
         b2 _ktbitfsc                     @62      0
         ub2 _ktbitwrp                    @62      0x0000
      ub4 ktbitbas                        @64      0xf72537a5
   struct ktbbhitl[1], 24 bytes           @68
      struct ktbitxid, 8 bytes            @68
         ub2 kxidusn                      @68      0x0008
         ub2 kxidslt                      @70      0x0044
         ub4 kxidsqn                      @72      0x004a0e5e
      struct ktbituba, 8 bytes            @76
         ub4 kubadba                      @76      0x0080011d
         ub2 kubaseq                      @80      0xf935
         ub1 kubarec                      @82      0x01
      ub2 ktbitflg                        @84      0x8000 (KTBFCOM)
      union _ktbitun, 2 bytes             @86
         b2 _ktbitfsc                     @86      9
         ub2 _ktbitwrp                    @86      0x0009
      ub4 ktbitbas                        @88      0xf7251d0b


BBED> p ktbbh.ktbbhitl[0].ktbitflg
ub2 ktbitflg                              @60      0x2001 (KTBFUPB)
```

从上面可以看到，ktbbh itl[0]对应的事务状态当前是 0x2001 (KTBFUPB)，说明此事务一直没有提交，状态不对，需要修改，代码如下：

```
BBED> modify /x 0x8000 ktbbh.ktbbhitl[0].ktbitflg
 File: C:\oracle\ora92\datadb\SYSTEM01.DBF (0)
 Block: 62              Offsets:   60 to  571           Dba:0x00000000
------------------------------------------------------------------------
 80000000 a53725f7 08004400 5e0e4a00 1d018000 35f90100 00800900 0b1d25f7
 0001d200 ffffb601 ff047f06 7f060000 d2005f3f 173fdb3e 9a3e5a3e 1a3eda3d
```

```
 993d583d 183dd73c 9b3c5c3c 1d3cdd3b a13b643b 233be33a a73a593a 1d3adc39
 9b395839 1739da38 99385d38 1d38e137 a0375937 ff04d536 8d364d36 0c36cb35
 8e354635 0a35bc34 7d344034 0334c233 81334133 ff32bf32 83324332 0832cb31
 8a314b31 0831c530 89304d30 1030cd2f 8c2f472f 022fc62e 842e3e2e fd2dbe2d
 7e2d3d2d fd2cbd2c 812c432c 022cbc2b 6c2b1c2b dc2a992a 5d2a212a dc299729
 52290d29 cd288d28 4d280d28 cd278c27 49270a27 ce269126 0726c725 89254125
 fe24b524 73243124 eb23a523 5f231923 d7229422 52220a22 be217721 3021eb20
 a5205e20 1820d21f 901f4c1f 051fbc1e 731e2e1e e81da21d 591d101d c61c7c1c
 321ce81b a21b5d1b 151bd01a 8b1a4a1a 0a1ac619 86194619 f218b218 72182f18
 e7179b17 50170117 c4168216 4016fb15 b2156115 1e15d714 91144714 fd13af13
 6c132913 ea12ab12 68122612 d1117b11 3511ef10 b0105b10 0510c20f 7f0f3a0f
 f30ea80e 620e180e d30d8a0d 440dfa0c b40c6a0c 270ce00b 990b550b 0d0b840a
 010ac309 7e093809 f608b508 74083308 ef07ac07 69070000 00000000 00000000
 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

 <32 bytes per line>

BBED> p ktbbh.ktbbhitl[0].ktbitflg
ub2 ktbitflg                                @60       0x0080 (NONE)
```

修改完成后，如果发现有错误，可以使用 undo 命令回退，如下所示：

```
BBED> undo
BBED> modify /x 0120 filename 'C:\oracle\ora92\datadb\SYSTEM01.DBF' block 62. of
fset 60.
 File: C:\oracle\ora92\datadb\SYSTEM01.DBF (0)
 Block: 62               Offsets:   60 to  571           Dba:0x00000000
------------------------------------------------------------------------
 01200000 a53725f7 08004400 5e0e4a00 1d018000 35f90100 00800900 0b1d25f7
 0001d200 ffffb601 ff047f06 7f060000 d2005f3f 173fdb3e 9a3e5a3e 1a3eda3d
 993d583d 183dd73c 9b3c5c3c 1d3cdd3b a13b643b 233be33a a73a593a 1d3adc39
 9b395839 1739da38 99385d38 1d38e137 a0375937 ff04d536 8d364d36 0c36cb35
 8e354635 0a35bc34 7d344034 0334c233 81334133 ff32bf32 83324332 0832cb31
 8a314b31 0831c530 89304d30 1030cd2f 8c2f472f 022fc62e 842e3e2e fd2dbe2d
 7e2d3d2d fd2cbd2c 812c432c 022cbc2b 6c2b1c2b dc2a992a 5d2a212a dc299729
 52290d29 cd288d28 4d280d28 cd278c27 49270a27 ce269126 0726c725 89254125
 fe24b524 73243124 eb23a523 5f231923 d7229422 52220a22 be217721 3021eb20
 a5205e20 1820d21f 901f4c1f 051fbc1e 731e2e1e e81da21d 591d101d c61c7c1c
 321ce81b a21b5d1b 151bd01a 8b1a4a1a 0a1ac619 86194619 f218b218 72182f18
 e7179b17 50170117 c4168216 4016fb15 b2156115 1e15d714 91144714 fd13af13
 6c132913 ea12ab12 68122612 d1117b11 3511ef10 b0105b10 0510c20f 7f0f3a0f
 f30ea80e 620e180e d30d8a0d 440dfa0c b40c6a0c 270ce00b 990b550b 0d0b840a
 010ac309 7e093809 f608b508 74083308 ef07ac07 69070000 00000000 00000000
 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

 <32 bytes per line>
```

使用 undo 命令以后，事务状态又恢复到以前，如下：

```
BBED> p ktbbh.ktbbhitl[0].ktbitflg
ub2 ktbitflg                                @60       0x2001 (KTBFUPB)

BBED> modify /x 0x0080 ktbbh.ktbbhitl[0].ktbitflg
 File: C:\oracle\ora92\datadb\SYSTEM01.DBF (0)
 Block: 62               Offsets:   60 to  571           Dba:0x00000000
```

```
------------------------------------------------------------------------
 00800000 a53725f7 08004400 5e0e4a00 1d018000 35f90100 00800900 0b1d25f7
 0001d200 ffffb601 ff047f06 7f060000 d2005f3f 173fdb3e 9a3e5a3e 1a3eda3d
 993d583d 183dd73c 9b3c5c3c 1d3cdd3b a13b643b 233be33a a73a593a 1d3adc39
 9b395839 1739da38 99385d38 1d38e137 a0375937 ff04d536 8d364d36 0c36cb35
 8e354635 0a35bc34 7d344034 0334c233 81334133 ff32bf32 83324332 0832cb31
 8a314b31 0831c530 89304d30 1030cd2f 8c2f472f 022fc62e 842e3e2e fd2dbe2d
 7e2d3d2d fd2cbd2c 812c432c 022cbc2b 6c2b1c2b dc2a992a 5d2a212a dc299729
 52290d29 cd288d28 4d280d28 cd278c27 49270a27 ce269126 0726c725 89254125
 fe24b524 73243124 eb23a523 5f231923 d7229422 52220a22 be217721 3021eb20
 a5205e20 1820d21f 901f4c1f 051fbc1e 731e2e1e e81da21d 591d101d c61c7c1c
 321ce81b a21b5d1b 151bd01a 8b1a4a1a 0a1ac619 86194619 f218b218 72182f18
 e7179b17 50170117 c4168216 4016fb15 b2156115 1e15d714 91144714 fd13af13
 6c132913 ea12ab12 68122612 d1117b11 3511ef10 b0105b10 0510c20f 7f0f3a0f
 f30ea80e 620e180e d30d8a0d 440dfa0c b40c6a0c 270ce00b 990b550b 0d0b840a
 010ac309 7e093809 f608b508 74083308 ef07ac07 69070000 00000000 00000000
 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

 <32 bytes per line>

BBED> p ktbbh.ktbbhitl[0].ktbitflg
ub2 ktbitflg                                @60       0x8000 (KTBFCOM)
```

修改完成后，需要修改数据库的 checksum(checkval)的值，否则数据块的校验不对，Oracle 会把这个块作为坏块处理，代码如下：

```
BBED> sum apply
```

至此全部修改完成。

优化小技巧　模拟 ORA-8102 及处理过程

今天老白在讲到 BBED 修改索引数据时只是点到为止，可能大家心里在骂老白不够厚道，实际上修改过程是十分烦琐的，大家如果不亲自去做一遍，是没法搞明白的。因此，这里通过一个模拟测试来说明今天处理 ORA-8102 的全过程。

通过 BBED，将 con$表中_NEXT_CONSTRAINT 那一行的 con#从 6001 改为 6000。改后重启数据库，然后操作：

```
SQL> conn scott/tiger
已连接。
SQL> CREATE table tcon1( a integer);

表已创建。

SQL> CREATE table tcon2( a integer not null);
CREATE table tcon2( a integer not null)
*
ERROR 位于第 1 行:
ORA-00604: 递归 SQL 层 1 出现错误
ORA-08102: 未找到索引关键字, obj# 49, dba 4277961 (2)
```

此时已经出现了类似的不一致。通过 Trace 文件发现：

```
*** 2008-08-08 16:56:58.984
*** SESSION ID:(12.7) 2008-08-08 16:56:58.984
oer 8102.2 - obj# 49, rdba: 0x004146c9
kdk key 8102.2:
  ncol: 1, len: 4
  key: (4):  03 c2 3d 05
```

表中的值是 c2 3d 05（6004），通过 Trace 文件查看索引的值：

```
BH (0x653EFC74) file#: 1 rdba: 0x004146c9 (1/83657) class 1 ba: 0x65273000

col 0; len 2; (2):  c2 3d
row#135[632] flag: -----, lock: 0, data:(6):  00 40 01 52 00 0c
col 0; len 3; (3):  c2 3d 02
```

索引中的值是 c2 3d 02（6001）。

下面需要将索引中的值设置为和表中一致，也就是 02 改为 05。

从上述计算：632+44+24×itl 槽位数（本案例是 2）=724。从 724 开始应该是：

```
00 00  00 40 01 52 00 0c 03 3d 02
```

因此 735 的数值是 02，可以通过 BBED 来确认一下 735 开始的数据是什么：

```
BBED>print 735
rowdata[26]
-----------
ub1 rowdata[26]                                    @735      0x02
```

我们看到，735 这个位置的值确实是我们所计算的 02，因此如果要解决这个问题，只要把 735 这个位置的 02 修改为 05 就可以了，启动 BBED，修改 735 的值，然后通到 sum apply 修改一下这个数据块的 checksum 值：

```
BBED>set file 1
BBED>set block 83657
BBED>modify /x 0x05 735
BBED>sum apply
```

这些操作完毕后，可以重启数据库，然后测试一下建表：

```
SQL> create table tcon2( a integer not null);
表已创建。
```

可以看出，使用 BBED 已经成功解决了问题。

优化小技巧 如何计算数据块中某个字段的位置

对于数据块来说，每一行的地址数据可以通过如下方式获得：

```
buffer tsn: 0 rdba: 0x00400152 (1/338)
scn: 0x0000.0150bef5 seq: 0x01 flg: 0x06 tail: 0xbef50601
frmt: 0x02 chkval: 0xad32 type: 0x06=trans data
Block header dump:  0x00400152
 Object id on Block? Y
 seg/obj: 0x1c  csc: 0x00.150bef4  itc: 1  flg: -  typ: 1 - DATA
```

```
     fsl: 0  fnx: 0x0 ver: 0x01

 Itl           Xid                  Uba         Flag  Lck        Scn/Fsc
0x01   0x0004.004.00006a8b  0x008001b1.070e.03  --U-    1  fsc 0x0000.0150bef5

data_block_dump,data header at 0x3513044
===============
tsiz: 0xfb8
hsiz: 0x16c
pbl: 0x03513044
bdba: 0x00400152
     76543210
flag=--------
ntab=1
nrow=173
frre=-1
fsbo=0x16c
fseo=0x232
avsp=0x192
tosp=0x192
0xe:pti[0] nrow=173   offs=0
0x12:pri[0]offs=0xfa7
0x14:pri[1]offs=0xf95
0x16:pri[2]offs=0xf84
0x18:pri[3]offs=0xf70
0x1a:pri[4]offs=0xf5c
0x1c:pri[5]offs=0xf48
0x1e:pri[6]offs=0xf36
0x20:pri[7]offs=0xf24
0x22:pri[8]offs=0xf10
0x24:pri[9]offs=0xefc
0x26:pri[10] offs=0xeea
0x28:pri[11] offs=0xed8
0x2a:pri[12] offs=0x232

tab 0, row 12, @0x232
tl: 26 fb: --H-FL-- lb: 0x1  cc: 3
col  0: [ 1]  80
col  1: [16]  5f 4e 45 58 54 5f 43 4f 4e 53 54 52 41 49 4e 54
col  2: [ 3]  c2 3d 02
```

从上面可以看到，第 13 行的位置是 0x232，这是一个相对地址，需要加上一个 BASE。BASE 的计算如下所示：

- 对于字典管理表空间，其 BASE 为 44+itc×24；
- 对于本地管理表空间，其 BASE 为 52+itc×24；

其中 itc 是 ITL 槽的数量，也就是转储文件里 itc 字段的值（上面加粗部分），因此本案例中的地址是 0x232+44+1×24=630。其中 630 指的是这一行行头的位置，即行头应该以 0x2c 开始：

```
BBED> set file 1
        FILE#           1

BBED> set block 338
```

```
        BLOCK#          338

BBED> print 630
rowdata[0]
----------
ub1 rowdata[0]
                            @630      0x2c

BBED> dump /v
 File: D:\Oracle\ORADATA\ORA92\SYSTEM01.DBF (1)
 Block: 338     Offsets:  630 to 1141  Dba:0x00400152
-------------------------------------------------------
 2c010301 80105f4e 4558545f 434f4e53 l ,....._NEXT_CONS
 54524149 4e5403c2 3d062c00 03018010 l TRAINT..=.,.....
 5f4e4558 545f434f 4e535452 41494e54 l _NEXT_CONSTRAINT
 02c23d2c 00030180 105f4e45 58545f43 l =,....._NEXT_C
 4f4e5354 5241494e 5403c22c 3a2c0003 l ONSTRAINT..,:,..
 0180105f 4e455854 5f434f4e 53545241 l ..._NEXT_CONSTRA
 494e5402 c23d2c00 03018010 5f4e4558 l INT..=,....._NEX
 545f434f 4e535452 41494e54 03c22c39 l T_CONSTRAINT..,9
 2c000301 80105f4e 4558545f 434f4e53 l ,....._NEXT_CONS
 54524149 4e5402c2 2c2c0003 0180105f l TRAINT..,,....._
 4e455854 5f434f4e 53545241 494e5403 l NEXT_CONSTRAINT.
 c22b642c 00030180 105f4e45 58545f43 l .+d,....._NEXT_C
 4f4e5354 5241494e 5402c22b 2c000301 l ONSTRAINT..+,...
 80105f4e 4558545f 434f4e53 54524149 l .._NEXT_CONSTRAI
 4e5403c2 2a642c00 0301800a 5359535f l NT..*d,.....SYS_
 43303031 373303c2 024a2c00 0301800a l C00173...J,.....
 5359535f 43303031 373103c2 02482c00 l SYS_C00171...H,.
 0301800a 5359535f 43303031 363503c2 l ....SYS_C00165..
 02422c00 0301800a 5359535f 43303031 l .B,.....SYS_C001
 363603c2 02432c00 0301800a 5359535f l 66...C,.....SYS_
 43303031 363703c2 02442c00 0301800a l C00167...D,.....
 5359535f 43303031 363803c2 02452c00 l SYS_C00168...E,.
 0301800a 5359535f 43303031 363903c2 l ....SYS_C00169..
 02462c00 0301800a 5359535f 43303031 l .F,.....SYS_C001
 373003c2 02472c00 0301800a 5359535f l 70...G,.....SYS_
 43303031 363303c2 02402c00 0301800a l C00163...@,.....
 5359535f 43303031 363403c2 02412c00 l SYS_C00164...A,.
 0301800a 5359535f 43303031 353603c2 l ....SYS_C00156..
 02392c00 0301800a 5359535f 43303031 l .9,.....SYS_C001
 353703c2 023a2c00 0301800a 5359535f l 57...:,.....SYS_
 43303031 353803c2 023b2c00 0301800a l C00158...;,.....
 5359535f 43303031 353903c2 023c2c00 l SYS_C00159...<,.
```

2c 01 03 这 3 个字节是行头，2c 表示这是一个普通的数据行，01 表示该行数据使用 1 号 ITL，03 表示这行有 3 个字段的值，01 80 表示第一个字段长度是一个字节，80 是 Oracle 数值类型的内部表示形式，表示 0（在 con$表中，这个字段是 owner 的 ID，有经验的 DBA 一眼就可以看出 owner=0 表示的是 SYS 用户）。紧随其后的是约束的名字，第一个字节 0x10 是十进制的 16，表示这个字段有 16 个字节：。

```
10 5f 4e 45 58 54 5f 43 4f 4e 53 l ,....._NEXT_CONS
```

```
 54 52 41 49 4e 54
```

以上 16+1 个字节就是_NEXT_CONSTRAINT，后面就是 con#的值：

```
03 c2 3d 06
```

03 是字段长度，c2 3d 06 表示一个数值。c2 表示小数点后面有 4 位，3d=61，61－1=60，06－1=05，因此这个数值是 6005。

优化小技巧　如何计算索引块中某个字段的位置

通过 Trace 文件中的索引叶节点信息，可得

```
Block header dump:  0x0200557a
 Object id on Block? Y
 seg/obj: 0x31  csc: 0x00.23049f4  itc: 5  flg: 0  typ: 2 - INDEX
     fsl: 0  fnx: 0x200557b ver: 0x01

 Itl           Xid                  Uba         Flag  Lck        Scn/Fsc
0x01   0x0002.003.00005edb  0x0080bcdc.0b96.01  CB--    0  scn 0x0000.021b9dd5
0x02   0x000a.012.000061e0  0x0080c84c.0b9d.15  --U-    4  fsc 0x003c.02304a14
0x03   0x0006.02f.000061f7  0x0080c767.0c47.20  --U-    5  fsc 0x004b.02304de1
0x04   0x0003.02a.00006239  0x0080c9b1.0bff.1c  C---    0  scn 0x0000.023049de
0x05   0x0007.000.000061ec  0x0080c8c4.0c59.0c  --U-    4  fsc 0x003c.023049f5

Leaf block dump
===============
header address 13835058056894570660=0xc0000000601b60a4
kdxcolev 0
KDXCOLEV Flags = - - -
kdxcolok 0
kdxcoopc 0x80: opcode=0: iot flags=--- is converted=Y
kdxconco 1
kdxcosdc 1
kdxconro 193
kdxcofbo 422=0x1a6
kdxcofeo 5114=0x13fa
kdxcoavs 5017
kdxlespl 0
kdxlende 13
kdxlenxt 0=0x0
kdxleprv 33576313=0x2005579
kdxledsz 6
kdxlebksz 7944


row#192[5166] flag: ------, lock: 0, len=13, data:(6):  00 40 00 aa 00 11
col 0; len 4; (4):  c3 45 42 0e
```

索引的位置计算也是 BASE+5166（44+24*5+5166）。

索引记录的格式为：<LCK> 00 +ROWID+索引字段。比如，本例的索引如下所示：

```
00 00 00 40 00 aa 00 11 04 c3 45 42 0e
```

优化小技巧 自己动手写一个 dul

今天老白用 BBED 修改了索引的值，这里涉及一个十分大的话题——Oracle 数据文件的物理结构。应广大读者的要求，老白决定将本节的“优化小技巧”扩展开来，向大家介绍数据文件和数据块结构方面的一些问题。对于有兴趣的朋友，完全可以根据这一章的内容，编写一个大家都很向往的 Oracle 的工具。不过老白在这里介绍的还是一些比较粗浅的理论，如果你想写出一个像 d.c.b.a 的 aul 这样的工具来，还是需要花费很多心血的。在本节也向 d.c.b.a 表示致敬。

1. bootstrap$和 kcvfh 结构

kcvfh 是文件头结构。每个数据库的系统表空间中都存在一个超级块，这个块位于文件 1 的 1 号块。我们首先以 10GB 为蓝本来看一下 kcvfh 结构，其代码如下：

```
struct kcvfh {

kcbh kcvfhbfh;      /* 标准块头，每个块都有的结构，20 个字节 */
kccfhg kcvfhhdr;
krdba kcvfhrdb;     /* 指向 bootstrap$的 RDBA 地址，这个是我们今天要关注的重点，这个值只有在
                       FILE# 1 才存在 */
kscn kcvfhcrs;      /* 文件创建的 SCN */
ub4 kcvfhcrt;       /* 文件创建的时间戳 */
ub4 kcvfhrlc;
kscn kcvfhrls;
ub4 kcvfhbti;
kscn kcvfhbsc;
ub2 kcvfhbth;

ub2 kcvfhsta;
ub4 kcvfhcpc;
ub4 kcvfhrts;
ub4 kcvfhccc;
kcvcpg kcvfhbcp;
ub4 kcvfhbhz;
kcvmxcd kcvfhxcd;
ktsn kcvfhtsn;
ub2 kcvfhtln;
ktsn_name_t kcvfhtnm;
…

}
```

下面我们用 BBED 来协助分析，具体代码如下：

```
BBED> set file 1
FILE# 1

BBED> set block 1
BLOCK# 1

BBED> map /v
File: /opt/oracle/oradata/orcl/system01.dbf (1)
```

```
Block: 1 Dba:0x00400001
------------------------------------------------------------
Data File Header

struct kcvfh, 676 bytes @0
struct kcvfhbfh, 20 bytes @0
struct kcvfhhdr, 76 bytes @20
ub4 kcvfhrdb @96
struct kcvfhcrs, 8 bytes @100
ub4 kcvfhcrt @108
ub4 kcvfhrlc @112
struct kcvfhrls, 8 bytes @116
ub4 kcvfhbti @124
struct kcvfhbsc, 8 bytes @128
ub2 kcvfhbth @136
ub2 kcvfhsta @138
struct kcvfhckp, 36 bytes @484
ub4 kcvfhcpc @140
ub4 kcvfhrts @144
ub4 kcvfhccc @148
struct kcvfhbcp, 36 bytes @152
ub4 kcvfhbhz @312
struct kcvfhxcd, 16 bytes @316
word kcvfhtsn @332
ub2 kcvfhtln @336
text kcvfhtnm[30] @338
ub4 kcvfhrfn @368
struct kcvfhrfs, 8 bytes @372
ub4 kcvfhrft @380
struct kcvfhafs, 8 bytes @384
ub4 kcvfhbbc @392
ub4 kcvfhncb @396
ub4 kcvfhmcb @400
ub4 kcvfhlcb @404
ub4 kcvfhbcs @408
ub2 kcvfhofb @412
ub2 kcvfhnfb @414
ub4 kcvfhprc @416
struct kcvfhprs, 8 bytes @420
struct kcvfhprfs, 8 bytes @428
ub4 kcvfhtrt @444

ub4 tailchk @8188

BBED> print kcvfhrdb
ub4 kcvfhrdb @96 0x00400179
```

从上面可以看出，bootstrap$对应的地址是 0x00400179（1/377）。下面我们来看看 1/377 的块转储数据：

```
Start dump data blocks tsn: 0 file#: 1 minblk 377 maxblk 385
buffer tsn: 0 rdba: 0x00400179 (1/377)
scn: 0x0000.0000014e seq: 0x01 flg: 0x04 tail: 0x014e1001
frmt: 0x02 chkval: 0xe733 type: 0x10=DATA SEGMENT HEADER - UNLIMITED
```

```
Hex dump of block: st=0, typ_found=1
Extent Control Header
-----------------------------------------------------------------
Extent Header:: spare1: 0 spare2: 0 #extents: 1 #blocks: 7
last map 0x00000000 #maps: 0 offset: 4128
Highwater:: 0x0040017d ext#: 0 blk#: 3 ext size: 7
#blocks in seg. hdr's freelists: 1
#blocks below: 3
mapblk 0x00000000 offset: 0
Unlocked
Map Header:: next 0x00000000 #extents: 1 obj#: 56 flag: 0x40000000
Extent Map
-----------------------------------------------------------------
0x0040017a length: 7

nfl = 1, nfb = 1 typ = 1 nxf = 0 ccnt = 3
SEG LST:: flg: USED lhd: 0x0040017c ltl: 0x0040017c
```

下面再来看看 obj#=56 是什么对象，具体代码如下：

```
SQL> SELECT name FROM sys.obj$ where obj#=56;
NAME
------------------------------
BOOTSTRAP$
```

对应的表是 bootstrap$，而 bootstrap$是个什么对象呢？下面我们来看看：

```
SQL> desc sys.bootstrap$
名称 是否为空? 类型
----------------------------------------- -------- ----------------------------
LINE# NOT NULL NUMBER
OBJ# NOT NULL NUMBER
SQL_TEXT NOT NULL VARCHAR2(4000)

SQL> SELECT * FROM sys.bootstrap$;

LINE# OBJ#
---------- ----------
SQL_TEXT
--------------------------------------------------------------------------------
-1 -1
8.0.0.0.0

0 0
CREATE ROLLBACK SEGMENT SYSTEM STORAGE ( INITIAL 112K NEXT 1024K MINEXTENTS 1
MAXEXTENTS 32765 OBJNO 0 EXTENTS (FILE 1 BLOCK 9))

20 20
CREATE TABLE ICOL$("OBJ#" NUMBER NOT NULL,"BO#" NUMBER NOT NULL,"COL#" NUMBER NOT

LINE# OBJ#
---------- ----------
SQL_TEXT
--------------------------------------------------------------------------------
NULL,"POS#" NUMBER NOT NULL,"SEGCOL#" NUMBER NOT NULL,"SEGCOLLENGTH" NUMBER NOT
```

```
NULL,"OFFSET" NUMBER NOT NULL,"INTCOL#" NUMBER NOT NULL,"SPARE1" NUMBER,"SPARE2"
NUMBER,"SPARE3" NUMBER,"SPARE4" VARCHAR2(1000),"SPARE5" VARCHAR2(1000),"SPARE6"
DATE) STORAGE ( OBJNO 20 TABNO 4) CLUSTER C_OBJ#(BO#)

40 40
CREATE INDEX I_ICOL1 ON ICOL$(OBJ#) PCTFREE 10 INITRANS 2 MAXTRANS 255 STORAGE
(INITIAL 64K NEXT 1024K MINEXTENTS 1 MAXEXTENTS 2147483645 PCTINCREASE 0 OBJNO
40 EXTENTS (FILE 1 BLOCK 249))

LINE# OBJ#
---------- ----------
SQL_TEXT
--------------------------------------------------------------------------------

28 28
CREATE TABLE CON$("OWNER#" NUMBER NOT NULL,"NAME" VARCHAR2(30) NOT NULL,"CON#"
NUMBER NOT NULL,"SPARE1" NUMBER,"SPARE2" NUMBER,"SPARE3" NUMBER,"SPARE4" VARCHAR2
(1000),"SPARE5" VARCHAR2(1000),"SPARE6" DATE) PCTFREE 10 PCTUSED 40 INITRANS 1
MAXTRANS 255 STORAGE ( INITIAL 64K NEXT 1024K MINEXTENTS 1 MAXEXTENTS 2147483645
PCTINCREASE 0 OBJNO 28 EXTENTS (FILE 1 BLOCK 169))

48 48

LINE# OBJ#
---------- ----------
SQL_TEXT
--------------------------------------------------------------------------------
CREATE UNIQUE INDEX I_CON1 ON CON$(OWNER#,NAME) PCTFREE 10 INITRANS 2 MAXTRANS 2
55 STORAGE ( INITIAL 64K NEXT 1024K MINEXTENTS 1 MAXEXTENTS 2147483645 PCTINCRE
ASE 0 OBJNO 48 EXTENTS (FILE 1 BLOCK 313))

49 49
CREATE UNIQUE INDEX I_CON2 ON CON$(CON#) PCTFREE 10 INITRANS 2 MAXTRANS 255 STORAGE
( INITIAL 64K NEXT 1024K MINEXTENTS 1 MAXEXTENTS 2147483645 PCTINCREASE 0
OBJNO 49 EXTENTS (FILE 1 BLOCK 321))


LINE# OBJ#
---------- ----------
SQL_TEXT
--------------------------------------------------------------------------------
15 15
CREATE TABLE UNDO$("US#" NUMBER NOT NULL,"NAME" VARCHAR2(30) NOT NULL,"USER#" NUMBER
NOT NULL,"FILE#" NUMBER NOT NULL,"BLOCK#" NUMBER NOT NULL,"SCNBAS" NUMBER,"
SCNWRP" NUMBER,"XACTSQN" NUMBER,"UNDOSQN" NUMBER,"INST#" NUMBER,"STATUS$" NUMBER
NOT NULL,"TS#" NUMBER,"UGRP#" NUMBER,"KEEP" NUMBER,"OPTIMAL" NUMBER,"FLAGS" NUMBER,
"SPARE1" NUMBER,"SPARE2" NUMBER,"SPARE3" NUMBER,"SPARE4" VARCHAR2(1000),"SPARE5"
VARCHAR2(1000),"SPARE6" DATE) PCTFREE 10 PCTUSED 40 INITRANS 1 MAXTRANS 255
STORAGE ( INITIAL 64K NEXT 1024K MINEXTENTS 1 MAXEXTENTS 2147483645 PCTINCREASE
0 OBJNO 15 EXTENTS (FILE 1 BLOCK 105))

LINE# OBJ#
---------- ----------
SQL_TEXT
```

```
--------------------------------------------------------------------------------

34 34
CREATE UNIQUE INDEX I_UNDO1 ON UNDO$(US#) PCTFREE 10 INITRANS 2 MAXTRANS 255 STORAGE
( INITIAL 64K NEXT 1024K MINEXTENTS 1 MAXEXTENTS 2147483645 PCTINCREASE 0
OBJNO 34 EXTENTS (FILE 1 BLOCK 201))

35 35
CREATE INDEX I_UNDO2 ON UNDO$(NAME) PCTFREE 10 INITRANS 2 MAXTRANS 255 STORAGE
(INITIAL 64K NEXT 1024K MINEXTENTS 1 MAXEXTENTS 2147483645 PCTINCREASE 0 OBJNO

LINE# OBJ#
---------- ----------
SQL_TEXT
--------------------------------------------------------------------------------
35 EXTENTS (FILE 1 BLOCK 209))

29 29
CREATE CLUSTER C_COBJ#("OBJ#" NUMBER) PCTFREE 0 PCTUSED 50 INITRANS 2 MAXTRANS 2
55 STORAGE ( INITIAL 56K NEXT 1024K MINEXTENTS 1 MAXEXTENTS 2147483645 PCTINCRE
ASE 0 OBJNO 29 EXTENTS (FILE 1 BLOCK 177)) SIZE 300

30 30
CREATE INDEX I_COBJ# ON CLUSTER C_COBJ# PCTFREE 10 INITRANS 2 MAXTRANS 255 STORAGE

LINE# OBJ#
---------- ----------
SQL_TEXT
--------------------------------------------------------------------------------
( INITIAL 64K NEXT 1024K MINEXTENTS 1 MAXEXTENTS 2147483645 PCTINCREASE 0 OBJNO
30 EXTENTS (FILE 1 BLOCK 185))

25 25
CREATE TABLE PROXY_ROLE_DATA$("CLIENT#" NUMBER NOT NULL,"PROXY#" NUMBER NOT NULL,
"ROLE#" NUMBER NOT NULL) PCTFREE 10 PCTUSED 40 INITRANS 1 MAXTRANS 255 STORAGE
( INITIAL 64K NEXT 1024K MINEXTENTS 1 MAXEXTENTS 2147483645 PCTINCREASE 0 OBJNO
25 EXTENTS (FILE 1 BLOCK 145))


LINE# OBJ#
---------- ----------
SQL_TEXT
--------------------------------------------------------------------------------
26 26
CREATE INDEX I_PROXY_ROLE_DATA$_1 ON PROXY_ROLE_DATA$(CLIENT#,PROXY#) PCTFREE 10
INITRANS 2 MAXTRANS 255 STORAGE ( INITIAL 64K NEXT 1024K MINEXTENTS 1 MAXEXTENTS
2147483645 PCTINCREASE 0 OBJNO 26 EXTENTS (FILE 1 BLOCK 153))

27 27
CREATE UNIQUE INDEX I_PROXY_ROLE_DATA$_2 ON PROXY_ROLE_DATA$(CLIENT#,PROXY#,ROLE#)
PCTFREE 10 INITRANS 2 MAXTRANS 255 STORAGE ( INITIAL 64K NEXT 1024K MINEXTENTS
1 MAXEXTENTS 2147483645 PCTINCREASE 0 OBJNO 27 EXTENTS (FILE 1 BLOCK 161))

LINE# OBJ#
```

```
---------- ----------
SQL_TEXT
--------------------------------------------------------------------------------

17 17
CREATE TABLE FILE$("FILE#" NUMBER NOT NULL,"STATUS$" NUMBER NOT NULL,"BLOCKS"
NUMBER NOT NULL,"TS#" NUMBER,"RELFILE#" NUMBER,"MAXEXTEND" NUMBER,"INC" NUMBER,
"CRSCNWRP" NUMBER,"CRSCNBAS" NUMBER,"OWNERINSTANCE" VARCHAR2(30),"SPARE1" NUMBER,
"SPARE2" NUMBER,"SPARE3" VARCHAR2(1000),"SPARE4" DATE) PCTFREE 10 PCTUSED 40 INITRANS
1 MAXTRANS 255 STORAGE ( INITIAL 64K NEXT 1024K MINEXTENTS 1 MAXEXTENTS 214
7483645 PCTINCREASE 0 OBJNO 17 EXTENTS (FILE 1 BLOCK 113))


LINE# OBJ#
---------- ----------
SQL_TEXT
--------------------------------------------------------------------------------
41 41
CREATE UNIQUE INDEX I_FILE1 ON FILE$(FILE#) PCTFREE 10 INITRANS 2 MAXTRANS 255
STORAGE ( INITIAL 64K NEXT 1024K MINEXTENTS 1 MAXEXTENTS 2147483645 PCTINCREASE
0 OBJNO 41 EXTENTS (FILE 1 BLOCK 257))

42 42
CREATE UNIQUE INDEX I_FILE2 ON FILE$(TS#,RELFILE#) PCTFREE 10 INITRANS 2 MAXTRANS
255 STORAGE ( INITIAL 64K NEXT 1024K MINEXTENTS 1 MAXEXTENTS 2147483645 PCTIN
CREASE 0 OBJNO 42 EXTENTS (FILE 1 BLOCK 265))

LINE# OBJ#
---------- ----------
SQL_TEXT
--------------------------------------------------------------------------------

13 13
CREATE TABLE UET$("SEGFILE#" NUMBER NOT NULL,"SEGBLOCK#" NUMBER NOT NULL,"EXT#"
NUMBER NOT NULL,"TS#" NUMBER NOT NULL,"FILE#" NUMBER NOT NULL,"BLOCK#" NUMBER
NOT NULL,"LENGTH" NUMBER NOT NULL) STORAGE ( OBJNO 13 TABNO 1) CLUSTER
C_FILE#_BLOCK#(TS#,SEGFILE#,SEGBLOCK#)

19 19
CREATE TABLE IND$("OBJ#" NUMBER NOT NULL,"DATAOBJ#" NUMBER,"TS#" NUMBER NOT NULL,

LINE# OBJ#
---------- ----------
SQL_TEXT
--------------------------------------------------------------------------------
"FILE#" NUMBER NOT NULL,"BLOCK#" NUMBER NOT NULL,"BO#" NUMBER NOT NULL,"INDMETH
OD#" NUMBER NOT NULL,"COLS" NUMBER NOT NULL,"PCTFREE$" NUMBER NOT NULL,"INITRANS"
NUMBER NOT NULL,"MAXTRANS" NUMBER NOT NULL,"PCTTHRES$" NUMBER,"TYPE#" NUMBER
NOT NULL,"FLAGS" NUMBER NOT NULL,"PROPERTY" NUMBER NOT NULL,"BLEVEL" NUMBER,"LEAFCNT"
NUMBER,"DISTKEY" NUMBER,"LBLKKEY" NUMBER,"DBLKKEY" NUMBER,"CLUFAC" NUMBER,
"ANALYZETIME" DATE,"SAMPLESIZE" NUMBER,"ROWCNT" NUMBER,"INTCOLS" NUMBER NOT NULL,
"DEGREE" NUMBER,"INSTANCES" NUMBER,"TRUNCCNT" NUMBER,"SPARE1" NUMBER,"SPARE2"
NUMBER,"SPARE3" NUMBER,"SPARE4" VARCHAR2(1000),"SPARE5" VARCHAR2(1000),"SPARE6"
DATE) STORAGE ( OBJNO 19 TABNO 3) CLUSTER C_OBJ#(BO#)
```

```
LINE# OBJ#
---------- ----------
SQL_TEXT
--------------------------------------------------------------------------------

39 39
CREATE UNIQUE INDEX I_IND1 ON IND$(OBJ#) PCTFREE 10 INITRANS 2 MAXTRANS 255 STORAGE
( INITIAL 64K NEXT 1024K MINEXTENTS 1 MAXEXTENTS 2147483645 PCTINCREASE 0
OBJNO 39 EXTENTS (FILE 1 BLOCK 241))

14 14
CREATE TABLE SEG$("FILE#" NUMBER NOT NULL,"BLOCK#" NUMBER NOT NULL,"TYPE#" NUMBER
NOT NULL,"TS#" NUMBER NOT NULL,"BLOCKS" NUMBER NOT NULL,"EXTENTS" NUMBER NOT

LINE# OBJ#
---------- ----------
SQL_TEXT
--------------------------------------------------------------------------------
NULL,"INIEXTS" NUMBER NOT NULL,"MINEXTS" NUMBER NOT NULL,"MAXEXTS" NUMBER NOT NULL,
"EXTSIZE" NUMBER NOT NULL,"EXTPCT" NUMBER NOT NULL,"USER#" NUMBER NOT NULL,"LISTS"
NUMBER,"GROUPS" NUMBER,"BITMAPRANGES" NUMBER NOT NULL,"CACHEHINT" NUMBER NOT
NULL,"SCANHINT" NUMBER NOT NULL,"HWMINCR" NUMBER NOT NULL,"SPARE1" NUMBER,"SPARE2"
NUMBER) STORAGE ( OBJNO 14 TABNO 2) CLUSTER C_FILE#_BLOCK#(TS#,FILE#,BLOCK#)

6 6
CREATE CLUSTER C_TS#("TS#" NUMBER) PCTFREE 10 PCTUSED 40 INITRANS 2 MAXTRANS 255

LINE# OBJ#
---------- ----------
SQL_TEXT
--------------------------------------------------------------------------------
STORAGE ( INITIAL 64K NEXT 1024K MINEXTENTS 1 MAXEXTENTS 2147483645 PCTINCREASE
0 OBJNO 6 EXTENTS (FILE 1 BLOCK 57))

7 7
CREATE INDEX I_TS# ON CLUSTER C_TS# PCTFREE 10 INITRANS 2 MAXTRANS 255 STORAGE
(INITIAL 64K NEXT 1024K MINEXTENTS 1 MAXEXTENTS 2147483645 PCTINCREASE 0 OBJNO
7 EXTENTS (FILE 1 BLOCK 65))

21 21

LINE# OBJ#
---------- ----------
SQL_TEXT
--------------------------------------------------------------------------------
CREATE TABLE COL$("OBJ#" NUMBER NOT NULL,"COL#" NUMBER NOT NULL,"SEGCOL#" NUMBER
NOT NULL,"SEGCOLLENGTH" NUMBER NOT NULL,"OFFSET" NUMBER NOT NULL,"NAME" VARCHAR
2(30) NOT NULL,"TYPE#" NUMBER NOT NULL,"LENGTH" NUMBER NOT NULL,"FIXEDSTORAGE"
NUMBER NOT NULL,"PRECISION#" NUMBER,"SCALE" NUMBER,"NULL$" NUMBER NOT NULL,
"DEFLENGTH" NUMBER,"DEFAULT$" LONG,"INTCOL#" NUMBER NOT NULL,"PROPERTY" NUMBER NOT
NULL,"CHARSETID" NUMBER,"CHARSETFORM" NUMBER,"SPARE1" NUMBER,"SPARE2" NUMBER,
"SPARE3" NUMBER,"SPARE4" VARCHAR2(1000),"SPARE5" VARCHAR2(1000),"SPARE6" DATE) STORAGE
( OBJNO 21 TABNO 5) CLUSTER C_OBJ#(OBJ#)
```

```
LINE# OBJ#
---------- ----------
SQL_TEXT
--------------------------------------------------------------------------------
45 45
CREATE UNIQUE INDEX I_COL1 ON COL$(OBJ#,NAME) PCTFREE 10 INITRANS 2 MAXTRANS 255
STORAGE ( INITIAL 32K NEXT 1024K MINEXTENTS 1 MAXEXTENTS 2147483645 PCTINCREASE
0 OBJNO 45 EXTENTS (FILE 1 BLOCK 289))

46 46
CREATE INDEX I_COL2 ON COL$(OBJ#,COL#) PCTFREE 10 INITRANS 2 MAXTRANS 255 STORAGE
( INITIAL 32K NEXT 1024K MINEXTENTS 1 MAXEXTENTS 2147483645 PCTINCREASE 0 OBJNO
46 EXTENTS (FILE 1 BLOCK 297))

LINE# OBJ#
---------- ----------
SQL_TEXT
--------------------------------------------------------------------------------

47 47
CREATE UNIQUE INDEX I_COL3 ON COL$(OBJ#,INTCOL#) PCTFREE 10 INITRANS 2 MAXTRANS
255 STORAGE ( INITIAL 32K NEXT 1024K MINEXTENTS 1 MAXEXTENTS 2147483645 PCTINCREASE
0 OBJNO 47 EXTENTS (FILE 1 BLOCK 305))

5 5
CREATE TABLE CLU$("OBJ#" NUMBER NOT NULL,"DATAOBJ#" NUMBER,"TS#" NUMBER NOT NULL,
"FILE#" NUMBER NOT NULL,"BLOCK#" NUMBER NOT NULL,"COLS" NUMBER NOT NULL,"PCTFREE$"

LINE# OBJ#
---------- ----------
SQL_TEXT
--------------------------------------------------------------------------------
NUMBER NOT NULL,"PCTUSED$" NUMBER NOT NULL,"INITRANS" NUMBER NOT NULL,"MAXTRANS"
NUMBER NOT NULL,"SIZE$" NUMBER,"HASHFUNC" VARCHAR2(30),"HASHKEYS" NUMBER,"FUNC"
NUMBER,"EXTIND" NUMBER,"FLAGS" NUMBER,"DEGREE" NUMBER,"INSTANCES" NUMBER,"AVGCHN"
NUMBER,"SPARE1" NUMBER,"SPARE2" NUMBER,"SPARE3" NUMBER,"SPARE4" NUMBER,"SPARE5"
VARCHAR2(1000),"SPARE6" VARCHAR2(1000),"SPARE7" DATE) STORAGE ( OBJNO 5
TABNO 2) CLUSTER C_OBJ#(OBJ#)

23 23
CREATE TABLE PROXY_DATA$("CLIENT#" NUMBER NOT NULL,"PROXY#" NUMBER NOT NULL,

LINE# OBJ#
---------- ----------
SQL_TEXT
--------------------------------------------------------------------------------
"CREDENTIAL_TYPE#" NUMBER NOT NULL,"CREDENTIAL_VERSION#" NUMBER NOT NULL,"CREDENTIAL
_MINOR#" NUMBER NOT NULL,"FLAGS" NUMBER NOT NULL) PCTFREE 10 PCTUSED 40 INITRANS
1 MAXTRANS 255 STORAGE ( INITIAL 64K NEXT 1024K MINEXTENTS 1 MAXEXTENTS 214748
3645 PCTINCREASE 0 OBJNO 23 EXTENTS (FILE 1 BLOCK 129))

24 24
```

```
CREATE UNIQUE INDEX I_PROXY_DATA$ ON PROXY_DATA$(CLIENT#,PROXY#) PCTFREE 10 INITRANS
2 MAXTRANS 255 STORAGE ( INITIAL 64K NEXT 1024K MINEXTENTS 1 MAXEXTENTS 21
47483645 PCTINCREASE 0 OBJNO 24 EXTENTS (FILE 1 BLOCK 137))

LINE# OBJ#
---------- ----------
SQL_TEXT
--------------------------------------------------------------------------------

16 16
CREATE TABLE TS$("TS#" NUMBER NOT NULL,"NAME" VARCHAR2(30) NOT NULL,"OWNER#" NUMBER
NOT NULL,"ONLINE$" NUMBER NOT NULL,"CONTENTS$" NUMBER NOT NULL,"UNDOFILE#"
NUMBER,"UNDOBLOCK#" NUMBER,"BLOCKSIZE" NUMBER NOT NULL,"INC#" NUMBER NOT NULL,"
SCNWRP" NUMBER,"SCNBAS" NUMBER,"DFLMINEXT" NUMBER NOT NULL,"DFLMAXEXT" NUMBER NOT
NULL,"DFLINIT" NUMBER NOT NULL,"DFLINCR" NUMBER NOT NULL,"DFLMINLEN" NUMBER NOT
NULL,"DFLEXTPCT" NUMBER NOT NULL,"DFLOGGING" NUMBER NOT NULL,"AFFSTRENGTH" NUMBER
NOT NULL,"BITMAPPED" NUMBER NOT NULL,"PLUGGED" NUMBER NOT NULL,"DIRECTALLOWED"

LINE# OBJ#
---------- ----------
SQL_TEXT
--------------------------------------------------------------------------------
NUMBER NOT NULL,"FLAGS" NUMBER NOT NULL,"PITRSCNWRP" NUMBER,"PITRSCNBAS" NUMBER,
"OWNERINSTANCE" VARCHAR2(30),"BACKUPOWNER" VARCHAR2(30),"GROUPNAME" VARCHAR2(30),
"SPARE1" NUMBER,"SPARE2" NUMBER,"SPARE3" VARCHAR2(1000),"SPARE4" DATE) STORAGE
( OBJNO 16 TABNO 2) CLUSTER C_TS#(TS#)

43 43
CREATE UNIQUE INDEX I_TS1 ON TS$(NAME) PCTFREE 10 INITRANS 2 MAXTRANS 255 STORAGE
( INITIAL 64K NEXT 1024K MINEXTENTS 1 MAXEXTENTS 2147483645 PCTINCREASE 0 OBJ
NO 43 EXTENTS (FILE 1 BLOCK 273))

LINE# OBJ#
---------- ----------
SQL_TEXT
--------------------------------------------------------------------------------

8 8
CREATE CLUSTER C_FILE#_BLOCK#("TS#" NUMBER,"SEGFILE#" NUMBER,"SEGBLOCK#" NUMBER)
PCTFREE 10 PCTUSED 40 INITRANS 2 MAXTRANS 255 STORAGE ( INITIAL 24K NEXT 1024K
MINEXTENTS 1 MAXEXTENTS 2147483645 PCTINCREASE 0 OBJNO 8 EXTENTS (FILE 1 BLOCK
73)) SIZE 225

9 9
CREATE INDEX I_FILE#_BLOCK# ON CLUSTER C_FILE#_BLOCK# PCTFREE 10 INITRANS 2 MAXTRANS

LINE# OBJ#
---------- ----------
SQL_TEXT
--------------------------------------------------------------------------------
255 STORAGE ( INITIAL 64K NEXT 1024K MINEXTENTS 1 MAXEXTENTS 2147483645 PCTINCREASE
0 OBJNO 9 EXTENTS (FILE 1 BLOCK 81))

10 10
```

```
CREATE CLUSTER C_USER#("USER#" NUMBER) PCTFREE 10 PCTUSED 40 INITRANS 2 MAXTRANS
255 STORAGE ( INITIAL 64K NEXT 1024K MINEXTENTS 1 MAXEXTENTS 2147483645 PCTINCREASE
0 OBJNO 10 EXTENTS (FILE 1 BLOCK 89)) SIZE 372

11 11

LINE# OBJ#
---------- ----------
SQL_TEXT
--------------------------------------------------------------------------------
CREATE INDEX I_USER# ON CLUSTER C_USER# PCTFREE 10 INITRANS 2 MAXTRANS 255 STORAGE
( INITIAL 64K NEXT 1024K MINEXTENTS 1 MAXEXTENTS 2147483645 PCTINCREASE 0 OBJNO
11 EXTENTS (FILE 1 BLOCK 97))

56 56
CREATE TABLE BOOTSTRAP$("LINE#" NUMBER NOT NULL,"OBJ#" NUMBER NOT NULL,"SQL_TEXT"
VARCHAR2(4000) NOT NULL) PCTFREE 10 PCTUSED 40 INITRANS 1 MAXTRANS 255 STORAGE
( INITIAL 56K NEXT 1024K MINEXTENTS 1 MAXEXTENTS 2147483645 PCTINCREASE 0 OBJNO
56 EXTENTS (FILE 1 BLOCK 377))

LINE# OBJ#
---------- ----------
SQL_TEXT
--------------------------------------------------------------------------------

12 12
CREATE TABLE FET$("TS#" NUMBER NOT NULL,"FILE#" NUMBER NOT NULL,"BLOCK#" NUMBER
NOT NULL,"LENGTH" NUMBER NOT NULL) STORAGE ( OBJNO 12 TABNO 1) CLUSTER C_TS#(TS#)

32 32
CREATE TABLE CCOL$("CON#" NUMBER NOT NULL,"OBJ#" NUMBER NOT NULL,"COL#" NUMBER NOT
NULL,"POS#" NUMBER,"INTCOL#" NUMBER NOT NULL,"SPARE1" NUMBER,"SPARE2" NUMBER,

LINE# OBJ#
---------- ----------
SQL_TEXT
--------------------------------------------------------------------------------
"SPARE3" NUMBER,"SPARE4" VARCHAR2(1000),"SPARE5" VARCHAR2(1000),"SPARE6" DATE)
STORAGE ( OBJNO 32 TABNO 2) CLUSTER C_COBJ#(OBJ#)

54 54
CREATE INDEX I_CCOL1 ON CCOL$(CON#,COL#) PCTFREE 10 INITRANS 2 MAXTRANS 255 STORAGE
( INITIAL 64K NEXT 1024K MINEXTENTS 1 MAXEXTENTS 2147483645 PCTINCREASE 0
OBJNO 54 EXTENTS (FILE 1 BLOCK 361))

55 55

LINE# OBJ#
---------- ----------
SQL_TEXT
--------------------------------------------------------------------------------
CREATE UNIQUE INDEX I_CCOL2 ON CCOL$(CON#,INTCOL#) PCTFREE 10 INITRANS 2 MAXTRANS
255 STORAGE ( INITIAL 64K NEXT 1024K MINEXTENTS 1 MAXEXTENTS 2147483645 PCTINCREASE
0 OBJNO 55 EXTENTS (FILE 1 BLOCK 369))
```

```
22 22
CREATE TABLE USER$("USER#" NUMBER NOT NULL,"NAME" VARCHAR2(30) NOT NULL,"TYPE#"
NUMBER NOT NULL,"PASSWORD" VARCHAR2(30),"DATATS#" NUMBER NOT NULL,"TEMPTS#" NUMBER
NOT NULL,"CTIME" DATE NOT NULL,"PTIME" DATE,"EXPTIME" DATE,"LTIME" DATE,"RESOURCE$"
NUMBER NOT NULL,"AUDIT$" VARCHAR2(38),"DEFROLE" NUMBER NOT NULL,"DEFGRP#"

LINE# OBJ#
---------- ----------
SQL_TEXT
--------------------------------------------------------------------------------
NUMBER,"DEFGRP_SEQ#" NUMBER,"ASTATUS" NUMBER NOT NULL,"LCOUNT" NUMBER NOT NULL,
"DEFSCHCLASS" VARCHAR2(30),"EXT_USERNAME" VARCHAR2(4000),"SPARE1" NUMBER,"SPARE2"
NUMBER,"SPARE3" NUMBER,"SPARE4" VARCHAR2(1000),"SPARE5" VARCHAR2(1000),"SPARE6"
DATE) STORAGE ( OBJNO 22 TABNO 1) CLUSTER C_USER#(USER#)

44 44
CREATE UNIQUE INDEX I_USER1 ON USER$(NAME) PCTFREE 10 INITRANS 2 MAXTRANS 255
STORAGE ( INITIAL 64K NEXT 1024K MINEXTENTS 1 MAXEXTENTS 2147483645 PCTINCREASE 0
OBJNO 44 EXTENTS (FILE 1 BLOCK 281))

LINE# OBJ#
---------- ----------
SQL_TEXT
--------------------------------------------------------------------------------

18 18
CREATE TABLE OBJ$("OBJ#" NUMBER NOT NULL,"DATAOBJ#" NUMBER,"OWNER#" NUMBER NOT
NULL,"NAME" VARCHAR2(30) NOT NULL,"NAMESPACE" NUMBER NOT NULL,"SUBNAME" VARCHAR2
(30),"TYPE#" NUMBER NOT NULL,"CTIME" DATE NOT NULL,"MTIME" DATE NOT NULL,"STIME"
DATE NOT NULL,"STATUS" NUMBER NOT NULL,"REMOTEOWNER" VARCHAR2(30),"LINKNAME" VAR
CHAR2(128),"FLAGS" NUMBER,"OID$" RAW(16),"SPARE1" NUMBER,"SPARE2" NUMBER,"SPARE3"
NUMBER,"SPARE4" VARCHAR2(1000),"SPARE5" VARCHAR2(1000),"SPARE6" DATE) PCTFREE
10 PCTUSED 40 INITRANS 1 MAXTRANS 255 STORAGE ( INITIAL 16K NEXT 1024K MINEXTENTS

LINE# OBJ#
---------- ----------
SQL_TEXT
--------------------------------------------------------------------------------
1 MAXEXTENTS 2147483645 PCTINCREASE 0 OBJNO 18 EXTENTS (FILE 1 BLOCK 121))

36 36
CREATE UNIQUE INDEX I_OBJ1 ON OBJ$(OBJ#) PCTFREE 10 INITRANS 2 MAXTRANS 255 STORAGE
( INITIAL 64K NEXT 1024K MINEXTENTS 1 MAXEXTENTS 2147483645 PCTINCREASE 0
OBJNO 36 EXTENTS (FILE 1 BLOCK 217))

37 37
CREATE UNIQUE INDEX I_OBJ2 ON OBJ$(OWNER#,NAME,NAMESPACE,REMOTEOWNER,LINKNAME,

LINE# OBJ#
---------- ----------
SQL_TEXT
--------------------------------------------------------------------------------
SUBNAME) PCTFREE 10 INITRANS 2 MAXTRANS 255 STORAGE ( INITIAL 16K NEXT 1024K MINEXTENTS
```

```
1 MAXEXTENTS 2147483645 PCTINCREASE 0 OBJNO 37 EXTENTS (FILE 1 BLOCK 225)
)

38 38
CREATE INDEX I_OBJ3 ON OBJ$(OID$) PCTFREE 10 INITRANS 2 MAXTRANS 255 STORAGE
(INITIAL 64K NEXT 1024K MINEXTENTS 1 MAXEXTENTS 2147483645 PCTINCREASE 0 OBJNO 38
EXTENTS (FILE 1 BLOCK 233))


LINE# OBJ#
---------- ----------
SQL_TEXT
--------------------------------------------------------------------------------
2 2
CREATE CLUSTER C_OBJ#("OBJ#" NUMBER) PCTFREE 5 PCTUSED 40 INITRANS 2 MAXTRANS 25
5 STORAGE ( INITIAL 136K NEXT 1024K MINEXTENTS 1 MAXEXTENTS 2147483645 PCTINCREASE
0 OBJNO 2 EXTENTS (FILE 1 BLOCK 25)) SIZE 800

3 3
CREATE INDEX I_OBJ# ON CLUSTER C_OBJ# PCTFREE 10 INITRANS 2 MAXTRANS 255 STORAGE
( INITIAL 64K NEXT 1024K MINEXTENTS 1 MAXEXTENTS 2147483645 PCTINCREASE 0 OBJNO
3 EXTENTS (FILE 1 BLOCK 49))

LINE# OBJ#
---------- ----------
SQL_TEXT
--------------------------------------------------------------------------------

4 4
CREATE TABLE TAB$("OBJ#" NUMBER NOT NULL,"DATAOBJ#" NUMBER,"TS#" NUMBER NOT NULL,
"FILE#" NUMBER NOT NULL,"BLOCK#" NUMBER NOT NULL,"BOBJ#" NUMBER,"TAB#" NUMBER,
"COLS" NUMBER NOT NULL,"CLUCOLS" NUMBER,"PCTFREE$" NUMBER NOT NULL,"PCTUSED$"
NUMBER NOT NULL,"INITRANS" NUMBER NOT NULL,"MAXTRANS" NUMBER NOT NULL,"FLAGS" NUMBER
NOT NULL,"AUDIT$" VARCHAR2(38) NOT NULL,"ROWCNT" NUMBER,"BLKCNT" NUMBER,"EMPCNT"
NUMBER,"AVGSPC" NUMBER,"CHNCNT" NUMBER,"AVGRLN" NUMBER,"AVGSPC_FLB" NUMBER,"FLBCNT"
NUMBER,"ANALYZETIME" DATE,"SAMPLESIZE" NUMBER,"DEGREE" NUMBER,"INSTANCES"

LINE# OBJ#
---------- ----------
SQL_TEXT
--------------------------------------------------------------------------------
NUMBER,"INTCOLS" NUMBER NOT NULL,"KERNELCOLS" NUMBER NOT NULL,"PROPERTY" NUMBER
NOT NULL,"TRIGFLAG" NUMBER,"SPARE1" NUMBER,"SPARE2" NUMBER,"SPARE3" NUMBER,"SPARE4"
VARCHAR2(1000),"SPARE5" VARCHAR2(1000),"SPARE6" DATE) STORAGE ( OBJNO 4 TA
BNO 1) CLUSTER C_OBJ#(OBJ#)

33 33
CREATE INDEX I_TAB1 ON TAB$(BOBJ#) PCTFREE 10 INITRANS 2 MAXTRANS 255 STORAGE
(INITIAL 64K NEXT 1024K MINEXTENTS 1 MAXEXTENTS 2147483645 PCTINCREASE 0 OBJNO 33
EXTENTS (FILE 1 BLOCK 193))

LINE# OBJ#
---------- ----------
SQL_TEXT
```

```
--------------------------------------------------------------------------------

31 31
CREATE TABLE CDEF$("CON#" NUMBER NOT NULL,"OBJ#" NUMBER NOT NULL,"COLS" NUMBER,
"TYPE#" NUMBER NOT NULL,"ROBJ#" NUMBER,"RCON#" NUMBER,"RRULES" VARCHAR2(3),"MATCH#"
NUMBER,"REFACT" NUMBER,"ENABLED" NUMBER,"CONDLENGTH" NUMBER,"CONDITION" LONG,
"INTCOLS" NUMBER,"MTIME" DATE,"DEFER" NUMBER,"SPARE1" NUMBER,"SPARE2" NUMBER,"SPARE3"
NUMBER,"SPARE4" VARCHAR2(1000),"SPARE5" VARCHAR2(1000),"SPARE6" DATE) STORAGE
( OBJNO 31 TABNO 1) CLUSTER C_COBJ#(OBJ#)


LINE# OBJ#
---------- ----------
SQL_TEXT
--------------------------------------------------------------------------------
50 50
CREATE UNIQUE INDEX I_CDEF1 ON CDEF$(CON#) PCTFREE 10 INITRANS 2 MAXTRANS 255
STORAGE ( INITIAL 64K NEXT 1024K MINEXTENTS 1 MAXEXTENTS 2147483645 PCTINCREASE 0
OBJNO 50 EXTENTS (FILE 1 BLOCK 329))

51 51
CREATE INDEX I_CDEF2 ON CDEF$(OBJ#) PCTFREE 10 INITRANS 2 MAXTRANS 255 STORAGE (
INITIAL 64K NEXT 1024K MINEXTENTS 1 MAXEXTENTS 2147483645 PCTINCREASE 0 OBJNO
51 EXTENTS (FILE 1 BLOCK 337))

LINE# OBJ#
---------- ----------
SQL_TEXT
--------------------------------------------------------------------------------

52 52
CREATE INDEX I_CDEF3 ON CDEF$(ROBJ#) PCTFREE 10 INITRANS 2 MAXTRANS 255 STORAGE
( INITIAL 64K NEXT 1024K MINEXTENTS 1 MAXEXTENTS 2147483645 PCTINCREASE 0 OBJNO
52 EXTENTS (FILE 1 BLOCK 345))

53 53
CREATE INDEX I_CDEF4 ON CDEF$(ENABLED) PCTFREE 10 INITRANS 2 MAXTRANS 255 STORAGE
( INITIAL 64K NEXT 1024K MINEXTENTS 1 MAXEXTENTS 2147483645 PCTINCREASE 0 OBJNO

LINE# OBJ#
---------- ----------
SQL_TEXT
--------------------------------------------------------------------------------
53 EXTENTS (FILE 1 BLOCK 353))
已选择 57 行。
```

bootstrap$里包含了数据库创建最主要的系统数据字典。我们通过 bootstrap$可以找到 obj$等数据字典表的 obj#以及段头的地址。什么是 bootstrap 呢？从计算机术语上来看：

```
Bootstrap is a technique for loading the first few instructions of a computer program
into active memory and then using them to bring in the rest of the program.
```

如果一个软件很复杂，那么在软件启动时需要把某些结构先载入内存，以便于访问其他的软件结构，包括继续启动软件以及完成软件的后续操作，这段代码一般我们就称 bootstrap。那

么对于 Oracle 数据库，bootstrap 指的是什么呢？

```
In Oracle, Bootstrap refers to loading of metadata (data dictionary) before we OPEN
the database.
Bootstrap objects are classified as the objects (tables / indexes / clusters) with the
object_id below 56 as bootstrap objects.
These objects are mandatory to bring up an instance, as this contains the most important
metadata of the database.
```

这段文字翻译成中文的意思就是 Oracle 的 bootstrap 的含义是在数据库打开之前装载元数据（数据字典数据）。一般来说，object_id 小于 56 的对象都属于 bootstrap 对象，包括表、索引和簇。这些对象对于实例启动十分关键。

通过 bootstrap 的概念我们了解到，要想去分析 Oracle 数据库的结构，首先需要通过 SUPER BLOCK 去找到 bootstrap$的位置，然后通过 bootstrap$中的数据字典定义，找出 obj$,C_OBJ$, TAB$和 COL$等主要数据字典表的存储信息，然后找到这些数据以建立数据字典，并通过数据字典，找到所需要分析的表的信息，从而取出这些数据。

今天我们分析数据时用的是 Oracle 10g 的数据结构。对于 Oracle 9i 和 Oracle 8i 的数据结构，我想大家已经学会了使用 BBED，那么通过 BBED 就很容易分析清楚了。

2. 数据块的结构

本节将通过一个实际例子介绍一下数据块的结构。数据块结构是 dul 的基础。这里介绍的是 type: 0x06=trans data 的数据块的结构（本文涉及的数据库版本为 Oracle 10.2.0.1）。下面是一个块的逻辑结构：

```
buffer tsn: 4 rdba: 0x0100a714 (4/42772)
scn: 0x0000.0025486e seq: 0x02 flg: 0x04 tail: 0x486e0602
frmt: 0x02 chkval: 0xddf1 type: 0x06=trans data
Block header dump: 0x0100a714
Object id on Block? Y
seg/obj: 0xcc5c csc: 0x00.25486d itc: 3 flg: E typ: 1 - DATA
brn: 0 bdba: 0x100a711 ver: 0x01 opc: 0
inc: 0 exflg: 0

Itl Xid Uba Flag Lck Scn/Fsc
0x01 0xffff.000.00000000 0x00000000.0000.00 C--- 0 scn 0x0000.0025486d
0x02 0x0000.000.00000000 0x00000000.0000.00 ---- 0 fsc 0x0000.00000000
0x03 0x0000.000.00000000 0x00000000.0000.00 ---- 0 fsc 0x0000.00000000

data_block_dump,data header at 0x8eb227c
===============
tsiz: 0x1f80
hsiz: 0x16
pbl: 0x08eb227c
bdba: 0x0100a714
76543210
flag=----
ntab=1 ----有 1 个表，如果是 CLUSTER，可能大于 1
nrow=2 -----有 2 行
frre=-1
```

```
fsbo=0x16 ----空闲空间的开始（44+NO_OF_ITLS*24+0X16） NO_OF_ITLS:itl 插槽的数量
fseo=0x1e70 ----空闲空间尾部 （44+NO_OF_ITLS*24 +0x1e70）
avsp=0x1e5a
tosp=0x1e5a
0xe:pti[0] nrow=2 offs=0
0x12:pri[0] offs=0x1f79 ----第一行的地址 0x1f79+（44+NO_OF_ITLS*24）
0x14:pri[1] offs=0x1e70 ----第二行的地址 0x1e70+（44+NO_OF_ITLS*24）
block_row_dump:
tab 0, row 0, @0x1f79 ---第一行
tl: 7 fb: --H-FL-- lb: 0x0 cc: 2
col 0: *NULL*
col 1: [ 2] c1 02 ---数值 1
tab 0, row 1, @0x1e70 ---第二行
tl: 263 fb: --H-FL-- lb: 0x0 cc: 2
col 0: [254]
31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35
36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 39 30
31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35
36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 39 30
31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35
36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 39 30
31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35
36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 39 30
31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35
36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 39 30
31 32 33 34
col 1: [ 2] c1 03 --数值 2
end_of_block_dump
```

下面通过这个数据块对应的裸文件来分析：

```
00000000h: 06 A2 00 00 14 A7 00 01 6E 48 25 00 00 00 02 04 ; .?..?.nH%.....
00000010h: F1 DD 00 00 01 00 00 00 5C CC 00 00 6D 48 25 00 ; 褫......\?.mH%.
00000020h: 00 00 00 00 03 00 32 00 11 A7 00 01 FF FF 00 00 ; ......2..?.    ..
00000030h: 00 00 00 00 00 00 00 00 00 00 00 00 00 80 00 00 ; .............€..
00000040h: 6D 48 25 00 00 00 00 00 00 00 00 00 00 00 00 00 ; mH%.............
00000050h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000060h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000070h: 00 00 00 00 00 00 00 00 00 00 00 00 00 01 02 00 ; ................
00000080h: FF FF 16 00 70 1E 5A 1E 5A 1E 00 00 02 00 79 1F ;     ..p.Z.Z.....y.
00000090h: 70 1E 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; p...............
000000a0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
000000b0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
000000c0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
```

上面我们看到的似乎杂乱无章的数据，实际上包含了十分高明的数据定义。在分析 Oracle 数据块结构时，我深深地被 Oracle 精巧的设计所震撼。这简直就是一种艺术。在下面的篇幅中，我将带领大家去体会这种叹为观止的艺术。

先看前 20 个字节：

```
06 A2 00 00 14 A7 00 01 6E 48 25 00 00 00 02 04
F1 DD 00 00
```

这 20 个字节对应的结构为 kcbh，该结构的定义如下：

```
typedef struct kcbh_ {
ub1 type_kcbh;
ub1 frmt_kcbh;
ub1 spare1_kcbh;
ub1 spare2_kcbh;
krdba rdba_kcbh;
ub4 bas_kcbh; /* SCN 的 BASE 部分*/
ub2 wrp_kcbh; /*SCN 的 WRAD 部分*/
ub1 seq_kcbh; /* 相同 SCN 下的 seq#*/
ub1 flg_kcbh;
ub2 chkval_kcbh;
ub2 spare3_kcbh;
} kcbh;
```

第 0 个字节为类型：06 表示是数据块。第 1 个字节是 frmt：一般是 0x02，这里是 0xa2，用掩码 0x0f 取出就是 0x02。第 4～7 个字节 14 A7 00 01 是 RDBA 0x100A714(4/42772)，就是这个块。第 8～13 个字节 6E 48 25 00 00 00 是 scn（前面 4 个是 BASE，后面是 WRAP）：0x254864.00。第 14 个字节 02 是 SEQ：SEQ=0x02。第 15 个字节 04 是标志：flag=0x04。第 16～17 个字节 F1 DD 是校验和：0xddf1。

正好和逻辑转储的数据完全吻合：

```
buffer tsn: 4 rdba: 0x0100a714 (4/42772)
scn: 0x0000.0025486e seq: 0x02 flg: 0x04 tail: 0x486e0602
frmt: 0x02 chkval: 0xddf1 type: 0x06=trans data
```

下面我们来看接下来的 24 个字节：

```
F1 DD 00 00 01 00 00 00 5C CC 00 00 6D 48 25 00
00 00 00 00 03 00 32 00 11 A7 00 01 FF FF 00 00
```

从第 20 个字节开始的 24 个字节对应的结构为 ktbbh，该结构的定义如下：

```
typedef struct ktbbh_ { /* 10.2 版本的块头结构*/
ub1 ktbbhtyp; /* 块类型*/
ub4 ktbbhsid;
kscn ktbbhcsc; /* 最后一次 cleanout 的 SCN */
b2 ktbbhict; /* itl 槽的数量，这个数据必须先和 0x00ff 做按位与操作，取其低位数据*/
ub1 ktbbhflg; /* 标志*/
ub1 ktbbhfsl; /* free space lock FSL*/
krdba ktbbhfnx; /* freelists 链指针 */
} ktbbh;
```

这个结构共占 20 个字节。第 20 个字节 01 表示数据块的类型为 1(对应逻辑转储的 typ=01)。第 24～27 个字节对应逻辑转储的 OBJECT ID，这里 5C CC 00 00 是对象 ID，对应的十六进制值是 0xcc5c（注意，这里有 UB4 的字节对齐问题，所以中间空了 3 个字节）。第 28～35 个字节中紧接着的 6 个字节 6D 48 25 00 00 00 是对应 SCN 的值，其值是 0x0025486d.0000。第 36～37 字节表示 ITL 插槽的数量，对应逻辑转储中的 itc，用 0x00ff 掩码后就是 ITL 插槽的数量，03 00 表明本数据块有 3 个 ITL 槽。第 38 个字节是标志：FLAG 32。第 39 个字节是 FSL，这里的值是 00。第 40～43 个字节是 KDBA，表示下一个空闲的数据块的地址。11 A7 00 01 就是十六进制的 0x100a711。

从第 44 个字节开始是 ITL 表，每个 ITL 记录的长度为 24 个字节。可以看到，这个块有 3 个 ITL 槽，因此 ITL 占 72 个字节。下面我们开始分析。

```
F1 DD 00 00 01 00 00 00 5C CC 00 00 6D 48 25 00
00 00 00 00 03 00 32 00 11 A7 00 01 FF FF 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 80 00 00
6D 48 25 00
```

对应的是 ktbit 结构：

```
struct ktbit {
kxid ktbitxid; /* 事务号 */
kuba ktbituba; /* 最后一次修改 undo 地址 */
b2 ktbitflg; /* 块中锁的数量 */
ktbitun_t _ktbitun;
ub4 ktbitbas; /* SCN 的 BASE 部分内容 */
}
```

ktbit 结构是从第 44～51 的 8 个字节，其内容如下所示。

- UB2：UNDO 段 NUMBER。
- UB2：SLOT NUMBER。
- UB4：WRAP。

第 44～51 个字节是 XID(8 字节)：FF FF 00 00 00 00 00 00,就是十六进制的 0xffff.0000.00000000。第 52～59 个字节是 UBA（8 字节）：00 00 00 00 00 00 00 00，就是十六进制的 0x00000000.0000.00。第 60～61 个字节是 FLAG（2 字节）：00 80，表示十六进制的 0x0800。第 62～67 个字节是 SCN（6 字节）：00 00 6D 48 25 00，是十六进制的 0x00254864.00。

下面我们换一个数据块来继续介绍结构，这个数据块的逻辑转储如下所示：

```
tsiz: 0x1f98
hsiz: 0x14
pbl: 0x0cebec64
bdba: 0x0100028e
     76543210
flag=--------
ntab=1
nrow=1
frre=-1
fsbo=0x14
fseo=0x1f8e
avsp=0x1f7a
tosp=0x1f7a
0xe:pti[0]      nrow=1  offs=0
0x12:pri[0]     offs=0x1f8e
block_row_dump:
tab 0, row 0, @0x1f8e
tl: 10 fb: --H-FL-- lb: 0x1  cc: 2
col  0: [ 3]  31 31 31
col  1: [ 2]  c1 03
end_of_block_dump
```

这个块有 2 个 ITL 槽，今后所有的位置都要根据这个来计算。下面是一个重要的结构 kdbh：

```
struct kdbh {
ub1 kdbhflag;      /* 标志*/
ktno kdbhntab;     /* 块中有几张表的数据*/
ub2 kdbhnrow;      /* 行数据表中的记录数*/
sb2 kdbhfree;      /* 第一个空闲行的位置*/
sb2 kdbhfsbo;      /* 块中空闲地址的起始偏移量*/
sb2 kdbhfseo;      /* 块中空闲地区尾部的偏移量*/
b2 kdbhavsp;       /* 块中可用的空闲空间，单位为字节*/
b2 kdbhtosp;       /* 块中总的可用空间*/
}
```

kdbh 结构的位置的计算方法如下所示。

- 对于 ASSM（自动段空间管理）：76+（itls−1）× 24。
- 对于 MSSM（手工段空间管理）：68+（itls−1）× 24。

本例使用的是 ASSM，并且有 2 个 ITL 槽，因此基准偏数量是 76+（2−1）×24=100。下面我们来看这部分数据：

```
00 00 00 00 00 01 01 00 FF FF 14 00 8E 1F 7A 1F
7a 1f 00 00 01 00 8e 1f
```

它对应的逻辑结构转储如下：

```
100:FLAG 0(--------------)
101:NTAB 1
102-103:NROW 01 00: (nrow=1)
104-105:frre ff ff (-1)
106-107:fsbo 14 00:0x14
108-109:fseo 8e 1f:0x1f8e
110-111:avsp 7a 1f: 0x1f7a
112-113:tosp 7a 1f: 0x1f7a
```

紧跟着的数据结构是 kdbt 结构，是一个数组，在一个数据块中，每个表有一行记录（对于 CLUSTER 表，可能有多行，对于普通表，只有一行），数据结构如下：

```
struct kdbt {
b2 kdbtoffs; /* kdbpri 结构中的偏移量*/
b2 kdbtnrow; /* 记录数量*/
}
```

下面的数据就是这个例子中的：

```
114-115: 00 00:0x0000
116-117: 01 00 :0x0001 (这个块有 1 行数据)
```

由于这张表不是 CLUSTER 表，所以 kdbpri 结构的偏移量是 0，而且这个数据块中，这张表只有一条记录。下面紧接着是 kdbr 结构。该结构是一个数组，是 sb2 类型（sb2 相当于 short，2 个字节长）的，指向每一行的头。这里只有 1 行，指向 0x1f8e。对于 ASSM，加上 76 就是这个记录的位置：0x1f8e+76+24=8178(0x1FF2)。找出下面这些数据：

```
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
04 00 2c 01 02 03 31 31 31 02 c1 03
```

其中 2c 是行的开头，属于 kdrh 结构：

```
ub1 kdrhflag; /* 行状态标志
ub1 kdrhlock; /* itl 指针*/
ub1 kdrhccnt; /* 该行数据的字段数量*/
```

2c 是记录头的标志。

01：ITL 槽的索引（锁所在的 ITL 槽的编号）。

02：字段的数量，0x02 表示这个记录有 2 个字段。

后面是第一个字段的数据：

```
03 31 31 31
```

03 表示字段长度，后面 3 个 31 就是字符串'111'：

```
02 c1 03
```

第二个字段是 Number 类型，长度是 2（02），C1 03 是数值 2 的内部表示。大家可以看到这个数据块里只有一条记录，这条记录是在这个块的尾部，数据块里的数据是从底部向上增长的。紧跟在后面这个块的最后 4 个字节是块尾，块尾的 4 个字节的组成分为两部分，其中前两个字节分别是这个数据块的 seq 和 typ，在这个例子里是：

```
01 06
```

其中 01 是 seq，06 是数据块的 typ，也就是块的类别，06 表示数据块。数据块的最后两个字节是 scn base 的低 4 个字节：0x7763。这个块的 SCN 是 0x005f7763。我们可以看出，块尾的这 3 部分数据都是块头中有的。Oracle 为什么要设计这么一个看似重复的机制呢？实际上，Oracle 的一个数据块的大小和操作系统物理 I/O 的块大小是不同的，因此一个数据块的读取可能在多个不同的物理 I/O 中完成，因此就有可能产生数据读取的不一致性，Oracle 使用块尾来确保数据块读取的一致性。

要注意的是，这个例子是最简单的。如果字段的长度超过 253 个字节，那么 8 位是表示不了的，这时，会使用前导字符 FE。比如，FE 03 06 表示该字段的长度是 0x603，就是 1 539 个字节。要注意的是，在数据块里面没有字段类型，因此如果系统表空间损坏，需要导出数据，就必须了解表的结构，否则需要根据扫描的结果进行猜测。

3. 写 dul 的第一步

刚才老白和大家讨论了数据块的物理结构和逻辑结构之间的关系。实际上根据刚才的讨论结果，有一定编程经验的 DBA 应该具备了写一个简单 dul 的条件了。是不是有些人已经跃跃欲试了？那我们现在就开始吧。首先编写一个简单的程序，简要分析一下数据文件的某一个数据块，代码如下：

```
int main(argc,argv)
int   argc;
char *argv[];
{
   int BLOCKSIZE=8192;   /* 这必须等于 db_block_size */
```

```
   char                  *filename;                          /* 数据文件名*/
   int                   orablocks; /* 数据文件中包含的数据块的数量*/
   int                   i;                                 /* 计数器*/
   char                  blockbuffer[68000];
   int                  fd;
   struct                stat fileinfo;
   int                   blockid;
   char *block_header;
   if(argc<3)
     {
     printf("usage :blockcheck <filename> <blocksize> <block#>\n");
        exit(-1);

     }

   filename=argv[1];
   if(filename==NULL)
     {
     printf("usage :blockcheck <filename> <blocksize> <block#>\n");
        exit(-1);

     }
   BLOCKSIZE=atoi(argv[2]);
   if(BLOCKSIZE<=0)
     {
     printf("usage :blockcheck <filename> <blocksize> <block#>\n");
        exit(-1);
     }

   blockid=atoi(argv[3]);
   if(blockid<=0)
     {
     printf("usage :blockcheck <filename> <blocksize> <block#>\n");
        exit(-1);
     }
printf("bc-I-begin to check file[%s]-block[%d],blocksize[%d],
OFFSET[%d]\n",filename,blockid,BLOCKSIZE,OFFSET);
//打开数据文件
   if((fd=open(filename,O_RDONLY))<0)
                            exit(1);

   if(fstat(fd,&fileinfo )<0)
                            exit(1);
printf("blocksize=%d,OFFSET=%d\n",BLOCKSIZE,OFFSET);
    orablocks=fileinfo.st_size/(BLOCKSIZE-OFFSET);
printf("blocks=%d\n",orablocks);

#ifndef LINUX
    if(lseek(fd,OFFSET,SEEK_SET)<0)
      exit(-1);
#endif
    printf("bc-I-begin to allocate block_header buffer\n");
    block_header=malloc(BLOCKSIZE);
```

```
//定位数据块的位置
   if(lseek(fd,BLOCKSIZE*(blockid),SEEK_SET)<0)
                                   exit(1);
 //读出数据块
   if(read(fd,blockbuffer,BLOCKSIZE)<0)
                exit(1);
   printf("bc-I-read block[%d] success...\n",blockid);
   block_header=&blockbuffer[0];
//调用函数 print_bh 打印块头信息
   print_bh(block_header);
   return(0);
}
```

上面这段代码应该很容易读懂，其中调用的 print_bh 函数的内容如下：

```
#define FALSE      0
#define TRUE       1
#define NUMORABLOCK(fsize,orabsize) \
   ((fsize==(orabsize-OFFSET)) ? (1) : ((fsize/(orabsize-OFFSET))-1))

#define FILE_ID(dba) (dba >> 22)     /* 将 dba 右移 22 位以得到文件*/

#define BLOCK_ID(dba) (dba & 0x3fffff)/*取低位 22 位计算出数据块的块号*/

#define CHECKCONDITION(bh) ((bh->type==6) && (bh->flg & 0x4))
#define PRINT_HEADER(bh) fprintf(stdout,\
   "file# %d,block# %d \n type: 0x%x fmt: 0x%x rdba: 0x%x base SCN: 0x%x\
   \n wrap SCN: 0x%x chk: 0x%x",   BLOCK_ID(bh->rdba),\
   FILE_ID(bh->rdba),bh->type, bh->Fmt, bh->rdba, bh->bas, bh->wrp, bh->cks)

#define PRINT_COMPUTED_CHECKSUM(bh,cks) fprintf(stdout,\
   " [kcbhxor: 0x%x]\n\n",cks)
#define PRINT_FILEINFO(fname,fstat,orablocks) \
   fprintf(stdout,"name: %s size: %d n-blocks: %d\n",fname,\
   fstat.st_size,orablocks)
typedef struct _bh {
   ub1     type;         /* 类型                          */
   ub1     Fmt;          /* 格式                          */
   ub1     r1;           /* 保留字段                      */
   ub1     r2;           /* 保留字段                      */
   ub4     rdba;         /* rdba  数据块地址              */
   ub4     bas;          /* SCN 的 BASE 部分              */
   ub2     wrp;          /* SCN 的 WRAP 部分              */
   ub1     seq;          /* 相同 SCN 下的序号             */
   ub1     flg;          /* 块标志                        */
   ub2     cks;          /* checksum                      */
   ub1     r3;           /* 对齐使用*/
   ub1     r4;           /* 字对齐使用 */
} block_header;
typedef struct kxid { /* struct kxid */
  ub2                  kxidusn; /* undo  段的号*/
  ub2                  kxidslt; /* undo 事务槽*/
  ub4                  kxidsqn; /* wrap  */
} kxid_st;
```

```
typedef struct kuba { /*  struct kuba undo block 地址  */
  ub4                   kubadba; /* RDBA，块地址  */
  ub2                   kubaseq; /* 序号  */
  ub1                   kubarec; /* 块内记录号  */
} kuba_st;

typedef struct ktbbh
{
 ub1    ktbbhtyp;
 ub1    sp1;
 ub4    object_id;
 ub4    kscnbas;      /* SCN 的 BASE 部分  */
  ub2    kscnwrp;     /* SCN 的 WRAP 部分  */
  ub2    itlno_hi;
  ub1    itlno;
  ub1    flags;       /* 0x01:在 freelists 上。0x10:使用位图 */
  ub1    fsl;         /* 空闲空间锁*/
  ub4    krdba;       /*下一个空闲块的 RDBA*/
}ktbbh_st;

typedef struct kbh{
 block_header bh;
 ktbbh_st kbh_bbh;
 ktbit_st kbh_it;
} kbh_st;

void print_bh(kbh_st *buffer)
{
   printf(
     "file# %d,block# %d \n type: 0x%x fmt: 0x%x rdba: 0x%x base SCN: 0x%x \n wrap SCN: 
0x%x chk: 0x%x seq=0x%x flg=0x%x\n",
   FILE_ID(buffer->bh.rdba),
   BLOCK_ID(buffer->bh.rdba),
   buffer->bh.type, buffer->bh.Fmt,
   buffer->bh.rdba,buffer->bh.bas,
   buffer->bh.wrp, buffer->bh.cks,
   buffer->bh.seq,buffer->bh.flg)   ;

   printf(
    "Type:0x%x objid:%d scn:0x%x/0x%x itls:%d flags:0x%x fsl:0x%x next:
    0x%x(%d/%d)\n",
     buffer->kbh_bbh.ktbbhtyp,
     buffer->kbh_bbh.object_id,buffer->kbh_bbh.kscnbas,
     buffer->kbh_bbh.kscnwrp,buffer->kbh_bbh.itlno&0X00FF,
     buffer->kbh_bbh.flags,buffer->kbh_bbh.fsl,
     buffer->kbh_bbh.krdba,
     FILE_ID(buffer->kbh_bbh.krdba),
     BLOCK_ID(buffer->kbh_bbh.krdba));
   if(buffer->bh.Type==6)
   print_itl(block_header,buffer->kbh_bbh.itlno&0X00FF);

}
```

这里调用了 print_itl 函数，这个函数的内容如下：

```
void print_itl(char * buffer,int count)
{
 int i;
 ktbit_st *itls;
 char flg[5];
 itls=(ktbit_st *)(buffer+44);
 printf("\n Itl         Xid                 Uba            Flag  Lck       Scn/Fsc\n");
 for(i=0;i<count;i++)
 {
   get_flg(itls[i].ktbitflg,flg);
  printf("0x%03x    0x%04x.%03x.%08x  0x%08x.%04x.%02x    %s    %d  %s
0x%04x.%08x\n",i+1,itls[i].ktbitxid.kxidusn,

itls[i].ktbitxid.kxidslt,itls[i].ktbitxid.kxidsqn,itls[i].ktbituba.kubadba,itls[i].
ktbituba.kubaseq,itls[i].ktbituba.kubarec,
   flg,itls[i].ktbitflg&0x00ff,flg[0]=='C'?"scn":"fsc",itls[i].ktbiwrap,itls[i].
ktbitbas);
 }
}
```

其中 get_flg 的定义如下：

```
void get_flg(ub2 flag,char *flg)
{

 if(flag&0x8000)
   flg[0]='C';
 else
   flg[0]='-';
  if(flag&0x4000)
    flg[1]='B';
  else
   flg[1]='-';
  if(flag&0x2000)
   flg[2]='U';
  else
   flg[2]='-';
  if(flag&0x1000)
   flg[3]='T';
  else
   flg[3]='-';
  flg[4]='\0';

}
```

上面程序的执行结果如下：

```
bc-I-begin to check file[D:\ORACLE\ORADATA\ORCL\ORCL\USERS01.DBF]-block[40708],
blocksize[8192], OFFSET[0]
file id=3
blocksize=8192,OFFSET=0
blocksize=8192,OFFSET=0
blocks=82721
```

```
bc-I-begin to allocate block_header buffer
bc-I-read block[40708] success...
file# 4,block# 40708
 type: 0x6 fmt: 0xa2 rdba: 0x1009f04 base SCN: 0xfe622
 wrap SCN: 0x0 chk: 0x2882 seq=0x5 flg=0x6
Type:0x1 objid:51588 scn:0xfe5ab/0x0 itls:3 flags:0x0 fsl:0x32 next:0x1009f01(4/40705)

 Itl           Xid                  Uba              Flag  Lck        Scn/Fsc
0x001    0xffff.000.00000000  0x00000000.0000.00    C---    0  scn  0x0000.000fe5ab
0x002    0x0005.018.00000189  0x00801e2a.012c.22    --U-    4  fsc  0x0000.000fe622
0x003    0x0000.000.00000000  0x00000000.0000.00    ----    0  fsc  0x0000.00000000
```

本节中，我们完成了一个读取数据块的小程序。实际上，编写 dul 工具与编本节的这个小程序一样，只要我们按部就班地剖析数据文件和数据块，就完全可以将数据读取出来。

优化小技巧 bootstrap$的访问

有了上面的练习，我们就可以正式编写 dul 了。今天我们首先处理 1 号数据文件的超级块，从超级块获取 bootstrap$对象的信息，从而找出 obj$、tab$之类的对象的 obj#，代码如下所示：

```
void bootstrap()
{
 tsFiles TsFileList;
 int i;
 int FileCount;
 int fp;
 char Msg[80];
 ub4  *BootStrapDba;
 kcvfh *fh;
 int Ret;
 etmap ObjMap;
 blockListStruc BlockList[100000];
 int BlockCount;

 printTraceMsg(0,'I',"This is bootstrap function");
 FileCount=indexTsFiles(&TsFileList,0,ParameterList.db_block_size); // 查找所有的数据文件
 for(i=0;i<TsFileList.FileCount;i++)
 {
   printf("JUSDUL-I-0000,trace system file[%s]\n",FileList[TsFileList.TsPt[i]].FileName);
                              }
 if(FileCount<=0)
   {
     fatalError(FOP_ERR_OPENTSERR);
   }
 else
    printTraceMsg(0,'I',"Index system table space OK,Will going to do bootstrap...");
//打开 1 号文件，1 号文件肯定是 system 表空间的第一个文件
   fp=open(FileList[TsFileList.TsPt[0]].FileName,O_RDONLY);
   if(fp<0)
```

```
        {
          sprintf(Msg,"Can not open system tablespace to do
bootstrap[%s]",FileList[TsFileList.TsPt[0]].FileName);
          printTraceMsg(-9999,'F',Msg);
          exit(-1);
        }
//定位到文件头
    if(lseek(fp,ParameterList.db_block_size,SEEK_SET)<0)
        {
          sprintf(Msg,"Can not lseek system tablespace to do
bootstrap[%s]",FileList[TsFileList.TsPt[0]].FileName);
          printTraceMsg(-9999,'F',Msg);
          exit(-1);
        }
//读出文件的第一个块
    if(read(fp,BlockBuffer,ParameterList.db_block_size)<=0)
        {
          sprintf(Msg,"Can not read system tablespace to do
bootstrap[%s]",FileList[TsFileList.TsPt[0]].FileName);
          printTraceMsg(-9999,'F',Msg);
          exit(-1);
        }
//将 kcvfh*类型的 fh 指向读出缓冲区的头，fh->kcvfhrdb 就是超级块的地址
    fh=(kcvfh *)&(BlockBuffer[0]);
    printf("JUSDUL-I-0000,BootStrap block address[0x%x]\n",fh->kcvfhrdb);
//定位文件到超级块所在的数据块
    if(lseek(fp,ParameterList.db_block_size*BLOCK_ID(fh->kcvfhrdb),SEEK_SET)<0)
        {
          sprintf(Msg,"Can not lseek system tablespace 2 to do
bootstrap[%s]",FileList[TsFileList.TsPt[0]].FileName);
          printTraceMsg(-9999,'F',Msg);
          exit(-1);
        }
//读出超级块
    if(read(fp,BlockBuffer,ParameterList.db_block_size)<=0)
        {
          sprintf(Msg,"Can not read system tablespace 2 to do
bootstrap[%s]",FileList[TsFileList.TsPt[0]].FileName);
          printTraceMsg(-9999,'F',Msg);
          exit(-1);
        }
//超级块的 108 字节就是 bootstrap$的 RDBA
    BootStrapDba=(ub4 *)&(BlockBuffer[108]);

    printf("bootstrapdba=0x%08x\n",*BootStrapDba);
//取出 bootstrap$的数据
    unloadBootStrap(fp,*BootStrapDba,FileList[TsFileList.TsPt[0]].FileName);
//关闭文件，完成第一阶段的 bootstrap$的下载工作
    close(fp);
}
```

unloadBootStrap 是从数据文件中将 bootstrap$表的数据读出来，写到文件中，其代码如下：

```
int unloadBootStrap(int fvp,int BootStrapDba,char *FileName)
{
  colStruc TypeList;
  colStruc ClusterTypeList;
  char Msg[80];
  ktbbh *bh;
  FILE *fp,*fpDel;
  int Ret;
//打开 bootstrap.Data，将读出的数据写入到这个文件中
  fp=fopen("bootstrap.data","w+");
  if(fp==NULL)
    {
      sprintf(Msg,"Can not open bootstrap.data to write,ERROR[%d]",errno);
      printTraceMsg('F',FOP_ERR_CREATEFILE,Msg);
      exit(-1);
    }
//将 bootstrap$中状态为删除状态的行写入 bootstrap.del 中
//有时候读出刚刚被删除的行也是很有意义的
  fpDel=fopen("bootstrap.del","w+");
  if(fpDel==NULL)
    {
      sprintf(Msg,"Can not open bootstrap.del to write,ERROR[%d]",errno);
      printTraceMsg('F',FOP_ERR_CREATEFILE,Msg);
      exit(-1);
    }

  sprintf(Msg,"Open file ok,begin to unload bootstrap$");
  printTraceMsg('I',0,Msg);
//老白自己写的读记录的框架，由于时间仓促，比较简单，虽然不太规范，不过老白自己用的还可以
//设置列的数量以及每个列的类型
  TypeList.ColCount=3;
  TypeList.ColType[1]=TYPE_NUMBER;
  TypeList.ColType[2]=TYPE_NUMBER;
  TypeList.ColType[3]=TYPE_VARCHAR2;
//这个用于簇表访问，设置簇的键值字段
  ClusterTypeList.ColCount=0;

  printf("JUSDUL-I-0000,BootStrap block SegmentHdr address[0x%x]\n",BootStrapDba);
  printf("blockis
[%d],pos[%d]\n",BLOCK_ID(BootStrapDba),ParameterList.db_block_size*(BLOCK_ID
(BootStrapDba)));

  if(lseek(fvp,ParameterList.db_block_size*(BLOCK_ID(BootStrapDba)),SEEK_SET)<0)
    {
      sprintf(Msg,"Can not lseek system tablespace 3 to do
bootstrap[%s],errno[%d]",FileName,errno);
      printTraceMsg('F',-99,Msg);
      exit(-1);
    }
  if(read(fvp,BlockBuffer,ParameterList.db_block_size)<=0)
```

```
      {
        sprintf(Msg,"Can not read system tablespace 4 to do
bootstrap[%s],errno[%d]",FileName,errno);
        printTraceMsg('F',-99,Msg);
        exit(-1);
      }
//映射块头到 bh
   bh=(ktbbh *)&(BlockBuffer[20]);
//调用 unloadBlock 函数从一个数据块中读取数据
   Ret=unloadBlock(fp,fpDel,ParameterList.db_block_size,&TypeList,bh->ktbbhsid,
&ClusterTypeList,0);
//以下是重复的，读取下一个 bootstrap$块。老白为了图省事，对于 bootstrap$表没有从段头读取 Extent
//Map，然后通过 Extent Map 来找数据块。实际上 bootstrap$的内容也不会很多，所以就偷懒了。如果大家写
//Dul，千万不要这么偷懒

   if(lseek(fvp,ParameterList.db_block_size*(BLOCK_ID(BootStrapDba)+1),SEEK_SET)<0)
      {
        sprintf(Msg,"Can not lseek system tablespace 3 to do
bootstrap[%s],errno[%d]",FileName,errno);
        printTraceMsg('F',-99,Msg);
        exit(-1);
      }

      if(read(fvp,BlockBuffer,ParameterList.db_block_size)<=0)
      {
        sprintf(Msg,"Can not read system tablespace 5 to do
bootstrap[%s],errno[%d]",FileName,errno);
        printTraceMsg('F',-99,Msg);
        exit(-1);
      }
   Ret+=unloadBlock(fp,fpDel,ParameterList.db_block_size,&TypeList,bh->ktbbhsid,
&ClusterTypeList,0);
   ... ... 这里省略了 n 个类似的 unload
   fclose(fp);
   Ret=readObj();
   if(Ret<=0)
   {
     sprintf(Msg,"Can not get Obj$ information ,load failed");
     printTraceMsg('F',-99,Msg);
     exit(-1);
   }

}
```

优化小技巧 unloadBlock 函数

上一节老白没有介绍 unloadBlock 函数，而这个函数才是大家关心的焦点，因此本节将详细介绍这个函数。老白终究是 10 年没有写过 C 程序了，程序的结构确实有点惨不忍睹，不过意思到了就行了。大家记住是跟着老白学写 dul 而不是跟着老白学写 C 程序。

```
int unloadBlock(FILE *fp,FILE* fpDel,int BlockSize,colStruc *TypeList,int
DataObjectId,colStruc *ClusterTypeList,int TabIndex)
```

```
/**********************************************************************/
*fp:存储下载数据的文件指针
*fpDel:存储删除行的文件指针
*BlockSize:数据块的大小
*TypeList:字段列表
*DataObjectId：数据存储的 DataObj 值，我们扫描数据文件时只下载 ID 相同的
*ClusterTypeList：用于簇表下载的簇主键字段类型
*TabIndex：在簇表中的表号，如果不是簇表，就是 0
*
**********************************************************************/
{
  int itls;
  kcbh *bh;
  kdbt  *dbt1,*dbt2,*dbt;
  ub2 *pt;
  int BaseLine;
  char Msg[80];
  int i,j,k;
  ktbbh *bh1;
  kdbh *dbh;
  int Rows;
  int PreRows;
  unsigned char *RowPt;
  bh=(kcbh*)&(BlockBuffer[0]);
  Rows=0;
//如果类型不是 6，就不需要下载了，因为不是数据块
  if(bh->type_kcbh==6)
  {
    bh1=(ktbbh *) &(BlockBuffer[20]);
    BhInfo=*bh;
    BhDetail=*bh1;
    //如果数据块里的 dataobj 和我们下载的不一致，就不需要下载了，因为出错了
    if(bh1->ktbbhsid==DataObjectId)
      {
        CurrentRdba=bh->rdba_kcbh;
        sprintf(Msg,"Find a data block [0x%x](%d/%d)",bh->rdba_kcbh,
FILE_ID(bh->rdba_kcbh),BLOCK_ID(bh->rdba_kcbh));
//        printTraceMsg('I',0,Msg);
//取 ITL 的数量
        itls=bh1->ktbbhict&0x00ff;
//判断是不是 ASSM 的，如果是 ASSM 的，BASELINE 要比普通的多 8 个字节
        if(bh1->ktbbhflg &0x10)
          BaseLine=52;
        else BaseLine=44;
//取 kdbt 结构的地址，里面存储了每一行的位置

        dbt1=(kdbt*)&(BlockBuffer[BaseLine+14+itls*24]);
//取 kdbh 结构的地址，里面有关键的行数这个字段
           dbh=(kdbh*)&(BlockBuffer[BaseLine+itls*24]);
//如果 kdbhntab 大于 1，说明这个块里包含的表的数量大于 1，即这是一个簇
        if(dbh->kdbhntab>1)
          CLUSTER_TBL=true;
        else
          CLUSTER_TBL=false;
```

```
        if(CLUSTER_TBL==false){
//根据 kdbt 的结构找到每一行的地址，下载数据
          for(i=0;i<dbt1->kdbtnrow;i++)
            {
              CurrentRowNo=i;
          pt=(ub2*) &( BlockBuffer [BaseLine+14+4*dbh -> kdbhntab +itls*24+2*i]);
//某一行的指针被取到 RowPt 里
              RowPt=&(BlockBuffer[*pt+BaseLine+itls*24]);
              RowPoint=*pt;
//下载一行数据
              Rows+=unloadRow(fp,fpDel,RowPt,TypeList,i);

            }
          }
          else
          {
//如果是簇表，首先要下载簇主键

  unloadClusterKey(BlockSize,ClusterTypeList,dbh->kdbhntab,dbh,dbt1,BaseLine,itls);
          dbt=dbt1;
          PreRows=0;

          dbt2=(kdbt*)&(BlockBuffer[BaseLine+14+itls*24+4*TabIndex]);

          for(i=dbt2->kdbtoffs;i<dbt2->kdbtoffs+dbt2->kdbtnrow;i++)
            {
              pt=(ub2*)&(BlockBuffer[BaseLine+14+4*dbh->kdbhntab+itls*24+2*i]);
              CurrentRowNo=i;
              RowPt=&(BlockBuffer[*pt+BaseLine+itls*24]);
              RowPoint=*pt+BaseLine+itls*24;
              RowPoint1=*pt;

              Rows+=unloadClusterRow(fp,fpDel,RowPt,TypeList,i);

            }
          }

         return Rows;
      }
    else
      {
      return 0;
    }
  }
  else
  return 0;
}
```

优化小技巧　unloadRow 函数

在上一小节里已经从数据块计算出每一行的位置，然后通过 unloadRow 去下载一行数据。在这一节里，老白要把压箱底的 unloadRow 拿出来和大家分享了，该函数的代码如下：

```
int unloadRow(FILE* fp,FILE*fpDel,char * RowPt,colStruc * TypeList,int RowNum)
{
  char Msg[80];
  int i;
  char *pt;
  ub2 Length;
  ub2* Length2;
  ub1 Length1;
  kdrh * RowHd;
  ub1 itls;
  unsigned char buffer[65532];
  char ColBuffer[4000];
  RowHd=(kdrh *)RowPt;
  RowBuffer[0]='\0';
  CurrentRow=RowPt;
//如果行的第一个字节为 0，那么这一行没有数据，直接返回
  if(RowPt[0]==0)
  return 0;
//
  if((RowPt[0]&0x10)!=0)
  {
  //第一个字节为 0x3c 说明是删除的记录
        if(RowPt[0]==0x3c)
            {
              itls=(ub1 ) RowPt[1];
              if (itls>(BhDetail.ktbbhict&0x00ff))
                        {
                              fprintf(stderr,"bad row\n");
                return 0;
            }

            pt=RowPt+3;
            for(i=0;i<RowHd->kdrhccnt;i++)
            {
                        Length1=(ub1)pt[0];
                        if(Length1==0xff)
                        {
                              if(i==0)
                                strcat(RowBuffer,"\"\"");
                              else
                                  strcat(RowBuffer,",\"\"");
                    pt++;
                              continue;
                        }

              if(Length1!=0xfe)
                   {
                     Length=Length1;
                     pt++;
                   }
              else
                   {
                   Length2=(ub2*)&(pt[1]);
```

```
                    Length=*Length2;
                    pt=pt+3;
                    }
                memcpy(buffer,pt,Length);
                buffer[Length];
                if(i==0)
                    {
                        strcat(RowBuffer,"\"");

    strcat(RowBuffer,parseData(buffer,Length,TypeList->ColType[i+1]));
                    if(ParseSuc==0) return 0;
                    strcat(RowBuffer,"\"");
                    }
                else
                    {
                        strcat(RowBuffer,",\"");

    strcpy(RowBuffer,parseData(buffer,Length,TypeList->ColType[i+1]));
                    if(ParseSuc==0) return 0;

                    strcat(RowBuffer,"\"");
                    }
                pt=pt+Length;
                }
                fprintf(fpDel,"%s\n",RowBuffer);
               }
             else{
              }
             return 0;
        }
  //正常的记录是从 0x2c 开头的
    if((RowPt[0]!=0x2c))
    {
      return 0;
       if((RowPt[0]&0x20)==0)
        {
            return 0;
        }
       if((RowPt[0]&&0x03)!=0)
        {
            return 0;
        }
        if(RowPt[0]!=0)
        exit(-1);
       }
  //通过 itls 指针来判断这一行是否是有效行，itls 必须小于等于 itc 的值
    itls=(ub1 ) RowPt[1];
    if (itls>(BhDetail.ktbbhict&0x00ff))
    {
      fprintf(stderr,"bad row\n");
      return 0;
    }

    if(RowPt[0]==0x2c)
```

```
  pt=RowPt+3;
  else
  pt=RowPt+9;
  for(i=0;i<RowHd->kdrhccnt;i++)
  {
//每个字段的第一个字节表示列长度，如果是FE，那么说明行长度大于254
//需要使用3个字节来表示列长度
//如果是FF，说明该记录已经迁移了，不在原来的位置
    Length1=(ub1)pt[0];
    if(Length1==0xff)
      {
        if(i==0)
          strcat(RowBuffer,"\"\"");
        else
          strcat(RowBuffer,",\"\"");
     pt++;
          continue;
      }

    if(Length1!=0xfe)
      {
        Length=Length1;
        pt++;
      }

    else
      {
        Length2=(ub2*)&(pt[1]);
        Length=*Length2;
        pt=pt+3;
      }
//将这个列的值复制到buffer里
    memcpy(buffer,pt,Length);
    buffer[Length]=0;
//通过parseData函数将内部格式转为字符串类型，写入文件中
    if(i==0)
      {
        strcat(RowBuffer,"\"");
        strcat(RowBuffer,parseData(buffer,Length,TypeList->ColType[i+1]));
        strcat(RowBuffer,"\"");
      }
    else
      {
        strcat(RowBuffer,",\"");
        strcat(RowBuffer,parseData(buffer,Length,TypeList->ColType[i+1]));
        strcat(RowBuffer,"\"");
      }
    pt=pt+Length;
  }
  fprintf(fp,"%s\n",RowBuffer);
  return 1;
}
```

至此老白介绍的dul编写告一段落。老白只是为大家领了一段路，而真正要写出一个十分优

秀的 dul 工具，绝对是要花费很多精力的。更多的经验，问问 d.c.b.a 可能收获会更大。

这里没有介绍的内容还有很多，比如 Extent Map 分析。因为我们要下载一张表，首先必须找到段头，然后找到段头里的 Extent Map，将这张表中所有高水位以下的扩展统统找出来，然后一个块一个块地下载下来。

另外关于字段内部格式的转换，在 DSI 401E 里面有很详细的介绍，我这里也就不做过多解释了。

5月29日 突破困局

周六的会议没有达到我们预期的效果，大家都有些沮丧，不过这样的结果预先也曾考虑到了，在这种情况下只能接受。堤外损失堤内补，开发商带来的负面影响只能用我们更加细致的准备来弥补了。因此，今天开始的工作将是十分关键的，我们将要把前几天提出的纲要性的东西变成一系列可操作的文字和脚本，也许这个项目的优化效果可能取决于这几天的工作。这也等于整个项目周期的提前，如果 6 月 6 日的优化能够解决 80%的性能问题，那么后续这段时间只要针对 SQL 进行优化，就可以实现优化目标了。

大家决定除了老熊继续做 SQL 优化外，其他人都投入到优化方案细化的工作中。而老熊近期的重点除了基于索引的 SQL 优化之外，还要继续和开发商扯皮，和他们一起尽快找出优化那个 12 张表关联查询的优化方案。

用户的服务器是 IBM P650，操作系统是 AIX 5.2。对于操作系统方面的调整，主要通过 `vmtune` 调整 `maxperm%`和 `maxclient%`。另外，pbuf 略微不足，也需要加。在最初的时候，我们也考虑过把操作系统升级到 AIX 5.3。因为 AIX 5.3 在性能上比 AIX 5.2 要好很多，但是从安全考虑，我们暂时不考虑升级操作系统，而仅在操作系统层面上进行优化。

存储方面的问题也较多，不过由于存储升级的可能性几乎为 0，所以优化存储的基础也不存在，只能通过减少 I/O 的数量来改进存储的总体性能了。开始时考虑过把在线重做日志文件独立出去的问题，但由于存储已经插满了，无法扩充硬盘而无法实施。

在数据库方面，除了需要加大数据库缓冲池、共享池、日志缓冲区以外，重做日记文件太小，需要从 100MB 加大为 500MB，同时每个实例增加几组重做日志组，使每个实例的重做日志组的数量达到 8 组。对于共享池，决定启用 `session_cached_cursors` 参数，同时将 `cursor_sharing` 从 `exact` 修改为 `similar`。

对于数据库对象方面的调整，主要集中在一些大表的分区上，三套系统各有五六张大表需要调整。在这些表中，最大的表有近 80GB，最小的也有几吉字节。表分区的原则，已经几次征求开发商的意见，他们都不肯提出自己的建议，而让我们自己决定，因此分区主键的选择完全依赖于技术分析，而没有一点业务方面的支持。

我们进行分区主键选择的方法是将和该表相关的主要 SQL 全部找出来，根据逻辑读和物理读排序，从这些 SQL 的 `WHERE` 条件以及表连接条件中查找效率最高的分区主键。这种方法说起来容易，实现起来往往比较困难，因为很多表的分区主键并不是那么明显，仅仅从 SQL 的 `WHERE`

条件很难做出十分准确的定位。在以往我们做过的优化项目中，分区主键的确定一般都是我们和开发商一起协商的。不过值得庆幸的是，通过 SQL 分析发现，这些表都有十分典型的分区主键。老肖马上着手建立评估模型，初步评估几张超大表。按照我们确定的分区主键进行分区后，绝大多数 Top SQL 的性能都得到了不同程度的提升，对该表的扫描性能也可以有大幅度的提升。

单表重定义为分区表的实施方法有很多种，通过在线重定义、导出导入等都是可行的。某些对于停机窗口要求很严的系统，必须使用在线重定义的方法来实施，但是 Oracle 自身的在线重定义存在一定的风险。对于几十个吉字节的表，我们一般不太采用。目前也有一些第三方产品支持表的在线重定义，这些工具一般来说先通过一个查询将表中的数据复制到一个地方，然后从重做日志中挖取与这个表相关的变化，并将其更改到目标表中，最后用很短的时间切断源表，并将最终的变化更改到目标表。这种机制可以做到最小的停机时间，不过这种工具往往十分昂贵。

我们比较幸运，由于分区表的数量不多，并且停机窗口允许，所以建议客户采用最为稳妥的方法。首先将原来的表重命名，然后创建一个新的表（分区表），然后使用 `INSERT /*+ APPEND*/` 的方式导入数据，然后再创建索引。这种方式一旦出现问题，回退起来十分快捷，只要做一个重命名就可以了。表重建后，可以等业务系统跑上一段时间，确定没有问题后，再删除旧表。

由于客户的系统内存十分有限，所以为了提高 DB Cache 的效率，启用 keep 池是十分必要的。不过使用 keep 池也碰到了一个难题，由于开发商不愿意提供热表，所以所有的热表和热索引都必须自己查找。一般来说，热表应该从业务角度来定义，开发商提供的热表才是最准确的，但是在目前这种情况下，只能通过分析 `x$bh` 来获取热表。Oracle 8i 引入了 `touch` 这个统计数据块热度的计数器。我们通过统计 `x$bh` 中某个对象的 TCH 值，找出那些 TCH 统计值很高并且较小的对象，作为放入 keep 池的候选对象。这个查找工作十分艰苦，因为 DB Cache 在不断变化，所以需要采集不同时间段的多个采样，然后从采样中进行分拣，最终找到适合放入 keep 池的对象。一般的观点是放入 keep 池的应该是一些基本静态的小表。因为如果访问量十分大，放入 keep 池会十分合适，不过并不是只有静态的小表才适合放入 keep 池。keep 池的作用是提高 DB Cache 的使用效率，减少 buffer busy waits，同时也能有效地减少 cache buffers chains 相关的闩锁竞争。具体哪些对象可以放入 keep 池并不是十分严格，原则上，热的对象（表和索引）都适合放入 keep 池，而并不一定要求是要静态的（当然静态的表放入 keep 池是最佳的）。由于 keep 池的大小有限，所以放入 keep 池的对象要进行十分严格的筛选。并且不能把过多的对象放入 keep 池，因为 keep 池中的对象都是十分热的，所以 keep 池出现抖动或者出现轻微的不命中，都会较大地影响系统的性能。至于放入 keep 池的对象的大小，也不能一刀切地进行划界。有些几百兆字节甚至更大的表，如果访问十分频繁，需要长期驻留内存，放入 keep 池也不是不可以。

查找放入 keep 池对象的工作原本有老肖在做，他也已经列出了一个清单。今天我和老肖一起讨论了一下，发现这么做还是有一些问题，最好能够结合系统的特点来查找，否则只是通过热度来查找工作量太大。我以前开发过计费账务系统，对电信的业务比较熟悉。电信总局的第一代计费账务系统的数据字典那本厚厚的灰皮书当时被我翻了无数遍，那些 ACCT、SERV、CUST 之类的表，我是一辈子都不会忘记的。想当年我们 7 个人居然能够开发出一套符合电信总局规范

的账务系统，并且通过了电信总局的软件测试。现在，一个账务系统动不动就需要上百人来开发，真是不可想象。和老肖商量了一下，他继续通过脚本查找热表，查到的热表清单提交给我，由我根据业务特点进行筛选。在筛选过程中，我还可以通过开发商的现场项目经理小王进行确认。因为通过最近这段时间的合作，我们也帮了小王几次忙，所以我们之间的沟通渠道已经比较畅通。

查找热块的方法其实很简单：通过 v$bh 视图就可以定位。自从 Oracle 8 引入了 TCH 这个统计项以后，查找热块的工作变得简单起来，而不用像以前那样去转储 LRU 链的信息。首先通过下面的脚本获取 TCH 比较高的对象：

```
Select decode(pd.bp_id,1,'KEEP',2,'RECYCLE',3,'DEFAULT',4,
'2K SUBCACHE', 5,'4K SUBCACHE',6,'8K SUBCACHE',7,
'16K SUBCACHE',8,'32K SUBCACHE','UNKNOWN') subcache,
  bh.object_name object_name,bh.blocks,tch
from x$kcbwds ds, x$kcbwbpd pd,
     (SELECT /*+ use_hash(x) */ set_ds,
        o.name object_name, count(*) BLOCKS,sum(tch)  tch
       FROM obj$ o, x$bh x
       WHERE o.dataobj# = x.obj
         AND x.state !=0 and o.owner# !=0
       GROUP BY set_ds,o.name) bh
WHERE ds.set_id >= pd.bp_lo_sid
  AND ds.set_id <= pd.bp_hi_sid
  AND pd.bp_size != 0
  AND ds.addr=bh.set_ds
  AND TCH>2000
ORDER BY subcache,object_name;

SUBCACHE     OBJECT_NAME                               BLOCKS        TCH
------------ ------------------------------ ---------- ----------
DEFAULT      ACCT_ITEM_AGGR_10611                          30       2742
DEFAULT      ACCT_ITEM_AGGR_10611                          28       2754
```

类似上面的 ACCT_ITEM_AGGR_10611 就可能是放入 keep 池的候选对象。不过并不是查出来的对象都是应该放入 keep 池的，ACCT_ITEM_AGGR_10611 表接近 800MB，因此该表不适合放入 keep 池。

这个工作十分枯燥，还好老肖很有耐性，他交给我的清单上面列出了详细的信息，哪怕是已经被初步认为不适合放入 keep 池的对象，也只是在上面做了注释，而没有简单地删除。

优化小技巧　查找"热的"缓冲区的常用脚本

优化热块是每个优化项目都会遇到的问题。实际上在无法得到开发部门足够的支持下，我们只能通过分析数据库缓冲区获得第一手资料。从 Oracle 8i 开始，Oracle 修改了热块的处理算法，引入了 tch 这个统计值，在 x$bh 中可以根据 tch 来判断该数据块是否为热块。下面几个脚本就是十分有用的通过数据缓冲区获得热块信息的脚本（要注意的是，由于数据块在 DB Cache 中是动态的，所以以 x$bh 中只能获得一些热块的线索，并不能十分准确地定位热块，最好的定位热块的方法还是通过业务角度分析为好）：

```
SELECT decode(pd.bp_id,1,'KEEP',2,'RECYCLE',3,'DEFAULT',4,
```

```
'2K SUBCACHE', 5,'4K SUBCACHE',6,'8K SUBCACHE',7,
'16K SUBCACHE',8,'32K SUBCACHE','UNKNOWN') subcache,
  bh.object_name object_name,bh.blocks,tch
FROM x$kcbwds ds, x$kcbwbpd pd,
     (SELECT /*+ use_hash(x) */ set_ds,
        o.name object_name, count(*) BLOCKS,sum(tch)  tch
       FROM obj$ o, x$bh x
       WHERE o.dataobj# = x.obj
         AND x.state !=0 and o.owner# !=0
       GROUP BY set_ds,o.name) bh
WHERE ds.set_id >= pd.bp_lo_sid
  AND ds.set_id <= pd.bp_hi_sid
  AND pd.bp_size != 0
  AND ds.addr=bh.set_ds
  AND TCH>2000
ORDER BY subcache,object_name;

SQL> column c1 heading "Object|Name"                 format a30 truncate
SQL> column c2 heading "Object|Type"                 format a12 truncate
SQL> column c3 heading "Number of|Blocks"            format 999,999,999,999
SQL> column c4 heading "Percentage|of object|data blocks|in Buffer" format 999
SQL>
SQL> break on report
SQL> compute sum of c3 on report
SQL> spool dbcache.lst;
SELECT
   object_name       c1,
   object_type       c2,
   num_blocks        c3,
   (num_blocks/decode(sum(blocks), 0, .001, sum(blocks)))*100 c4
  FROM
     (SELECT
       o.object_name    object_name,
       o.object_type    object_type,
       count(1)         num_blocks
      FROM
      dba_objects  o,
      v$bh         bh
     WHERE
      o.object_id  = bh.objd
     AND
      o.owner not in ('SYS','SYSTEM')
     GROUP BY
      o.object_name,
      o.object_type
     ORDER BY
      count(1) desc
     )  t1,
     dba_segments s
  WHERE
      s.segment_name = t1.object_name
   AND
      num_blocks > 10
```

```
   GROUP BY
      object_name,
      object_type,
      num_blocks
   ORDER BY
      num_blocks desc
   ;

                                                                  Percentage
                                                                   of object
Object                         Object              Number of data blocks
Name                           Type                    Blocks   in Buffer
------------------------------ ------------ ---------------- -----------
CALL_EVENT_10604               TABLE                      709           0
VALUE_ADDED_OTHER_EVENT_10610  TABLE                      705           0
CALL_EVENT_10603               TABLE                      691           0
VALUE_ADDED_OTHER_EVENT_10605  TABLE                      682           0
CALL_EVENT_10610               TABLE                      675           0
CALL_EVENT_10606               TABLE                      672           0
CALL_EVENT_10601               TABLE                      621           0
CALL_EVENT_10602               TABLE                      572           0
```

第二个脚本分析第一个脚本找到的热块，看看它们在 DB Cache 中有多少，并且平时有多大比例的数据块在缓冲区中。通过这两个脚本，就可以初步定位可能放入 keep 池中的对象了。不过最准确地确定 keep 对象还是从业务角度。技术分析是最后的办法，而不是推荐的方法。

优化小技巧　什么时候需要使用多缓冲

并不是在任何场合都需要使用多缓冲池。通过对系统 I/O 性能的评估来确定是否使用多缓冲池是比较科学的方法。分配多缓冲池的原则是把经常使用的小型对象（比如参照表、常用的索引等）放入 keep 池中，把随机访问的大型表放入 recycle 池中。下面的步骤可以定位某个对象占用缓冲池的大小。

(1) 找出对象的内部 ID，具体代码如下：

```
SELECT  data_object_id, object_type
         FROM    user_objects
 WHERE   object_name = '<segment_name>'
 AND object_type='<object_type>';
```

(2) 找出该对象占用的缓冲区的大小，具体代码如下：

```
SELECT count(*) buffers
FROM    x$bh
WHERE   obj = <data_object_id>;
```

需要注意的是，上面的脚本可以查找非分区的对象。对于分区对象，需要将所有分区的缓冲区大小累加。

(3) 查询出系统的缓冲区的总数。

对于早期的版本，如果使用 db_block_buffers 参数，可以使用下面的查询：

```
SELECT  value "total buffers"
```

```
FROM    v$parameter
WHERE   name = 'db_block_buffers';
```

对于 Oracle 9i 数据库，可以使用如下的查询：

```
SELECT name,VALUE/<DB_BLOCK_SIZE>  “TOTAL BUFFERS”
FROM V$PARAMETER
WHERE NAME IN
(‘db_cache_size’,’db_keep_cache_size’,’db_recycle_cache_size’);
```

其中<DB_BLOCK_SIZE>是数据库默认块大小。

计算占用率的公式如下：

占用率 = buffers（第(2)步）/ total buffers（第(3)步）

应该把哪些对象放入哪个缓冲池中是一个十分复杂的问题。对于一个对象来说，如果块获取接近于物理读，那么说明该对象中的数据基本上不重用，因此可以把该对象放入 recycle 池中。上面的例子虽然看起来很简单，但在实际应用中又不是那么简单。由于应用系统的复杂性，对某张表的访问方式可能是多样性的，有基于索引的扫描，也可能存在全表扫描或者区域扫描（比如统计和模糊查询），简单地区分是否可以把某个表放入 recycle 池的方法是不存在的。另外，什么样的对象是小的可以放入 keep 池的对象，这个问题同样复杂。有一些 Oracle 技术资料认为 100 块以下的对象可以认为是小的对象，也有资料说 500 块以下的对象可以称为小对象，其实在实际的应用中并不能仅仅从对象的大小来进行区分。在实际应用中，各种环境下缓冲池的大小是不同的。对于一个部门级环境来说，数据库缓冲区总的大小可能只有 300MB，而对于一个大型的企业级环境来说，可能会有几十吉字节的缓冲池，因此根据应用的情况来确定使用哪种缓冲池是比较科学的方法。下面是一些多缓冲池使用的经验（注意，只是经验，并不是原则）。

- 使用多缓冲池肯定对系统的 IO 优化会有所帮助。
- 在进行数据库设计时，不光要考虑表在缓冲池中的配置，也要考虑索引在缓冲池中的配置。
- 对于分区的对象，可以按照单个分区来考虑，同一个对象的不同分区可能需要放到不同的缓冲区中。
- 变动比较小的、经常访问的小型对象（记录数 1000 行或几千行），不添加索引，直接放入 keep 池中。
- 小于 default 池 10%大小的，至少引起 1%物理 I/O 的对象，可以考虑放入 keep 池。
- 对于在缓冲区中的块数超过 75%的对象，如果 keep 池足够，可以考虑将该对象放入 keep 池中。
- 对于使用率十分高的索引，如果 keep 池容量充足，可以放到 keep 池中。
- 大型的经常做随机访问的表可以放到 recycle 池中。
- 可以适当使用非默认块大小的表空间（Oracle 9i 以上）。

优化小技巧 AIX 操作系统优化要点

一般来说，AIX 操作系统针对 Oracle 数据库的优化并不复杂，主要集中在异步 I/O、虚拟内

存管理等方面。对于异步 I/O，首先要明白一点，针对裸设备和针对文件系统的优化是不同的。AIX 5L 对于裸设备使用异步 I/O 是内核支持的，因此不需要使用 AIO Server；而文件系统使用异步 I/O 时，则需要使用 AIO Server。通过以下命令可以检查异步 I/O 的设置：

```
LN-BILL-DB-JF || / # lsattr -El aio0
autoconfig available STATE to be configured at system restart True
fastpath   enable    State of fast path                       True
kprocprio  39        Server PRIORITY                          True
maxreqs    4096      Maximum number of REQUESTS               True
maxservers 40        MAXIMUM number of servers per cpu        True
minservers 10        MINIMUM number of servers                True
```

maxreqs 是最大 AIO 请求数量，其默认值是 4096，对于较大的系统来说，4096 可能不够。由于 Oracle 和 AIX 的 bug，当 maxreqs 不足时会出现类似下面的错误：

```
Sun Dec 16 21:27:34 2007
Errors in file /data1/oracle9/app/admin/sdh/bdump/sdh_dbw3_462948.trc:
ORA-27061: skgfospo: waiting for async I/Os failed
IBM AIX RISC System/6000 Error: 22: Invalid argument
Sun Dec 16 21:27:34 2007
DBW3: terminating instance due to error 27061
Instance terminated by DBW3, pid = 462948
```

严重时会导致宕机。这个 bug 在 Oracle 9.2 和 Oracle 10.2 中都有，因此建议在安装数据库时将 maxreqs 参数调整为 8192 或者更高。

参数 maxservers 和 minservers 仅适用于使用文件系统的环境。maxservers 是每个 CPU 最大的 AIO Server 的数量，而不是系统中 AIO Server 的总数。对于 I/O 比较高的系统，需要设置较大的 maxservers 参数。通过 pstat 命令，可以查看当前 AIO Server 的数量，如果已经接近或者达到了最大的 AIO Server 数量，那么就需要加大 maxservers 参数了：

```
LN-BILL-DB-JF || / # pstat -a|grep aios
 66 a    428a      1  428a     0     0     1  aioserver
 67 a    438c      1  438c     0     0     1  aioserver
 69 a    458c      1  458c     0     0     1  aioserver
 70 a    468c      1  468c     0     0     1  aioserver
 71 a    478e      1  478e     0     0     1  aioserver
 72 a    4890      1  4890     0     0     1  aioserver
 73 a    4992      1  4992     0     0     1  aioserver
 74 a    4a94      1  4a94     0     0     1  aioserver
 75 a    4b96      1  4b96     0     0     1  aioserver
 76 a    4c98      1  4c98     0     0     1  aioserver
```

对于虚拟内存管理来说，作为 Oracle 服务器使用的系统，是不需要多少文件缓冲的。文件缓冲不会提高 Oracle 数据库的性能，因为 Oracle 数据库本身有自己的 DB Cache。AIX 操作系统参数 maxperm%可以限制文件缓存占用物理内存的百分比。不过 AIX 操作系统默认的 maxperm%值是 80%，这是一个很高的值，并不适合于 Oracle 数据库服务器。因此在调整 AIX 操作系统时，建议将 maxperm%调整为 10%或者更小。

IBM 的工程师可能会说文件缓冲是可以释放的。一旦操作系统需要，就可以将文件缓冲的内存释放出来使用，且确实在 IBM 的相关技术资料中，都建议在 AIX 5.3 以后的版本中安装 Oracle

数据库时，设置 maxperm%为 90%～95%，同时设置 lru_file_repage 为 0。从原理上来看，这种设置不无道理，不过由于 Unix 的算法问题，这些被文件缓冲占用的内存的释放需要一个过程，释放这些内存往往伴随着 I/O 和换页。当物理内存十分小时，由于没有达到最低阈值，所以文件缓冲中的内存不会释放。而这个时候如果有一个比较消耗内存的操作发生，那么文件缓冲的内存需要释放，这样就会缠上大量的 I/O 和换页操作，在某个瞬间可能导致系统资源被过度消耗，引起短暂的性能瓶颈，严重时还可能产生一些预想不到的后果。我曾经碰到过一个客户，由于做了一个 FTP 操作，想把一个 40GB 的备份集复制到另外一台服务器上，FTP 刚刚开始不到半分钟，这个节点上的 RAC 实例就宕了。后来经检查发现，这个节点被 RAC 驱逐了。这是由于文件缓冲释放时产生了大量系统调用，CPU 和 I/O 资源都出现了不足，最终造成 RAC 之间的通信超时。被另外一个节点认为这个节点出现了故障，主动驱逐了这个故障节点。

对于 pbuf 的调整，主要取决于操作系统中裸设备访问是否引起了大量的 pbuf 等待。在别的操作系统下，一般来说查看 I/O 是否存在等待，可以看 vmstat 命令的 b 字段的值；对于 AIX 操作系统，如果使用文件系统，可以看 b 的值；而对于使用裸设备的系统，b 可能永远为 0。比如，下面的系统就是使用裸设备的：

```
LN-BILL-DB-JF || / # vmstat 1 10
System Configuration: lcpu=6 mem=12288MB
kthr     memory              page              faults        cpu
----- ----------- ------------------------ ------------ -----------
 r  b   avm    fre    re pi po fr sr cy    in    sy    cs us sy id wa
 2  1 1963449 565281  0  0  1 20 67  0   150 28237 20593 18  7 67  8
 1  0 1964106 564582  0  0  0  0  0  0 14039 47759 37195 16 24 53  7
 1  0 1963522 565166  0  0  0  0  0  0 13860 41577 33567 13 15 64  8
 2  0 1963803 564884  0  0  0  0  0  0 13195 39947 31459 12 14 67  7
 7  0 1964053 564634  0  0  0  0  0  0 14537 45937 36789 26 29 35 10
 5  0 1964071 564625  0  0  0  0  0  0 15405 48860 41174 22 25 46  7
 0  0 1963522 565164  0  0  0  0  0  0 13498 34095 32703 10 10 75  5
 1  0 1963846 564839  0  0  0  0  0  0 12381 46809 30043 15 20 57  7
 3  0 1963534 565151  0  0  6  0  0  0 15080 44522 38531 26 27 37 11
 2  0 1964185 564500  0  0  0  0  0  0 15801 41351 42505 20 20 50 10
```

如果使用 vmstat -I 命令，就可以看到另外的现象，具体如下：

```
root:/>vmstat -I 2 10
  kthr      memory              page              faults        cpu
-------- ----------- ------------------------ ------------ -----------
 r  b  p   avm    fre    fi fo  pi  po  fr  sr   in    sy    cs us sy id wa
 4  1 15 1529390 648297 31 63   0   0  63 336  199 26183  8006 42  9 36 13
 3  0 45 1531733 645953  0  0   0   0   0   0 4672 30147 10393 38 14  0 48
 7  0 38 1533274 644410  0  1   0   0   0   0 5430 39781 11837 52 16  0 33
 2  0 37 1531244 646439  0  0   0   0   0   0 4726 32458 10638 40 15  0 45
 5  0 38 1533318 644364  0  0   0   0   0   0 4096 22612  9455 37 11  0 52
 2  0 41 1530308 647373  0 64   0   1   0   0 4122 24930  9439 36 15  0 49
 3  0 40 1534883 642797  0  0   0   0   0   0 4179 29303  9481 46 13  0 41
 5  0 41 1536024 641654  0  0   0   0   0   0 4138 28142  9851 32 14  0 54
 3  0 41 1531396 646281  0  0   0   0   0   0 4414 30345 10438 42 14  0 44
 0  0 34 1532657 645019  0  0   0   0   0   0 5125 34884 12842 36 11  0 54
```

可以看到，b 的值相当高，这是由于裸设备的 I/O 访问存在较为严重的性能问题。这个时候使用 vmstat -v，可以看到一些和 I/O 相关的信息，具体如下：

```
root:/>vmstat -v
              2097152 memory pages
              1985190 lruable pages
               222013 free pages
                    2 memory pools
               136502 pinned pages
                 80.1 maxpin percentage
                  5.0 minperm percentage
                 12.0 maxperm percentage
                 11.9 numperm percentage
               237978 file pages
                  0.0 compressed percentage
                    0 compressed pages
                  0.0 numclient percentage
                 12.0 maxclient percentage
                    0 client pages
                    0 remote pageouts scheduled
              1393707 pending disk I/Os blocked with no pbuf
               288770 paging space I/Os blocked with no psbuf
                93511 filesystem I/Os blocked with no fsbuf
                    0 client filesystem I/Os blocked with no fsbuf
                    0 external pager filesystem I/Os blocked with no fsbuf
```

从上面的代码可以看到，由于缺乏 pbuf 而导致等待。在一天的业务高峰期开始和结束时各执行一次这个命令，就可以看出在业务高峰期由于 pbuf 不足而导致的等待事件有多少。如果这个等待增长十分快，那么就说明 pbuf 需要进行优化。要注意的是，pbuf 的调整需要十分小心。调整不当反而会降低系统性能，并且 pbuf 在 AIX 的不同版本的策略是不同的，调整方法也不同，调整之前一定要认真阅读 IBM 的相关手册。

5月30日 账务处理

今天是账期处理的第一天，老于建议我好好休息一下，晚点过去，他过去顶一下。其实我没有睡懒觉的习惯，早上还是 7：30 就起床了，到餐厅吃完早饭后又回到酒店，躺在床上看看“第一时间”节目。在深圳的时候，早上我一般会看看江川主持的“第一现场”，或者看看央视 2 套的“第一时间”。我准备中午的时候直接到北站那边吃午餐，那边 12 块钱的自助午餐还是不错的。

下午到办公室时老于正在帮客户优化一个统计程序。这个统计程序每个月的月底都要执行一次，随着数据量的增加，这个程序越跑越慢，上个月跑了 3 个多小时才出来，这回跑了将近 6 个小时才出来。老于看了一下，这是一个 7 张大表连接的查询，其中 4 张是超过 300MB 的大表，一张是 150MB 的表，一张是 50 多兆字节的远程表。整个 SQL 的执行计划十分复杂，还涉及 dblink，老于帮他们调整了一下 SQL，使用了一个临时表，结果两个小时就完成了统计。

我又看了一下，临时表确实起到了很好的作用。在原来的 SQL 里，临时表产生的结果集被多次使用，每次使用都要重新扫描一遍 ACCT 表，而这张表的大小超过 500MB，而生成的临时表数据只有几十兆字节。这种情况是典型的可以通过临时表进行优化的。

老于看到我来了，说：“老白，看看能不能再优化一点，客户还是嫌太慢。”我说老于你先回酒店休息一下吧，这个 SQL 我继续搞。老于走后，我看了一下，目前老于把一个 SQL 拆成了两个 SQL，由于减少了对 ACCT 这张大表的访问，效率提高了不少。不过对于临时表的访问还是有 10 次，临时表的大小虽然瘦身到了 1/10 的大小，不过还是有优化的余地。如果在临时表上创建一个索引，那么就不需要对临时表进行全表扫描了，而且这 10 个结果集是不会有重复记录的。如果把 `UNION` 改为 `UNION ALL`，就可以减少不必要的剔重操作了，所以我给客户以下的建议。

- 在临时表上创建索引，进一步提高临时表的访问效率。在将临时表插入数据之前删除索引，数据插入完毕后创建索引。
- 将多个结果集的 `UNION` 操作改为 `UNION ALL`。
- 通过 Hash 连接替代以前的嵌套循环。
- 在执行这个 SQL 前，加大 `PGA_AGGREGATE_TARGET` 的值，执行完毕后再将该参数改回原来的值。

调整 `PGA_AGGREGATE_TARGET` 值的目的是提高 Hash 连接的效率。由于这个模块一般在非营业时间执行，所以有足有的内存供系统使用，完全可以通过加大 `PGA_AGGREGATE_TARGET` 来增加 Hash 连接的缓冲区的大小。但是由于物理内存有限，在营业时间无法设置更高的 `PGA_AGGR-`

EGATE_TARGET，所以这个 SQL 执行完毕后，这个参数必须还原。还好 PGA_AGGREGATE_TARGET 这个参数是动态的，可以随时调整。

根据上面的思路调整后，我们再次执行了这个模块，刚才的调整起到了神奇的作用，不到 20 分钟，这个模块就跑完了。通知业务部门对生成的数据进行核对，结果和刚才做的完全一致。

其实 SQL 优化有时候就是这样，其中并没有特别高深的技术，每种优化技术都很简单，但是这些技术如何使用、如何组合才能达到最大的效果是十分关键的。水平高的人就是能够找到这些组合的人。比如说对于绝大多数统计程序来说，Hash 连接往往比嵌套循环的效率高，因为统计往往是对大量数据的汇总处理，索引的效率往往并不能决定整个程序的性能，而连接方式往往对性能的影响更大。在这个优化案例里，临时表的使用提高了数据的使用效率，从而使性能提升了 3 倍以上。UNION ALL 的使用减少了结果集剔重的开销，使性能提升了 10%，不过起到最大作用的是 Hash 连接的使用。Hash 连接的使用使两张大表的连接只通过对两张表进行一遍扫描就完成了，这样比嵌套循环的效率高了很多倍。

下午其实没有什么事情，今天是周末，只有做账务处理的人在现场，看着他们忙碌的样子，我们也帮不上什么忙。SQL 分析已经基本完成，这几天实际上是要看看账务处理过程中还有哪些 SQL 需要优化，其他的工作比较少。

晚上和老于就在故宫附近找了家李连贵熏肉大饼，要了瓶二锅头。二两二锅头下肚，老于发起了感慨。做 DBA 十多年了，越来越感觉信心不足，看看网上一批年轻人发的文章和博客，真有长江后浪推前浪的感觉，怕这么下去，用不了几年自己越来越没价值了。

确实，我们这批 DBA 大多数是从 20 世纪 90 年代初开始接触 Oracle 的，那时候参考书很少，甚至连 Metalink 账号都不太普及。这批 DBA 都是靠自己在工作中摸爬滚打积累经验的，理论基础往往都比较薄弱。老于感觉做了这么长时间 DBA，有些江郎才尽的感觉，真怕某天会被淘汰了。

我安慰老于说，DBA 是一个实践性很强的职业，虽然在某些理论方面，我们这批人没有年轻人研究的那么深，不过殊途同归。真正碰到问题，以我们的经验，我们也能做出正确的判断。解决问题，这十多年的经验积累也不是空中楼阁。有些人虽然理论基础十分好，不过如果缺乏实践经验，那么也很难把理论融入实践中，碰到问题照样无所适从。

现在 OTN 论坛上也经常有两大派的人在争吵，一派是经验派，基本上都是工作八九年以上的老 DBA，另外一派是学院派，理论基础十分扎实，基本上是最近三五年才开始从事 DBA 工作的人。经验派倚老卖老，学院派满嘴 INTERNAL，大家吵得不可开交。实际上作为 DBA，经验和理论都是必不可少的。随着年龄的增长，学习能力逐渐下降，再过几年，现在的学院派可能就会成为那时的经验派，经验增长了，理论反而荒疏了。

晚上回来后，老于提的问题让我思考了很多。这些年吃老本比较多，真正的学习比较少了，作为 DBA，知识更新是十分快的。而随着年龄的增长，学习能力越来越弱，早晚有一天，老于的忧虑也会成为我的忧虑。我们这批老 DBA 都比较怀旧，几年前我是坚决不接受 Oracle 9i 的，因为我一直认为 Oracle 8i 太完美了，直到有一天我做了一个 Oracle 9i 调优的项目，才开始真正

去认真研究 Oracle 9i，发现原来 Oracle 9i 更加完美。Oracle 10g 出来好长时间了，我还没认真看看 Oracle 10g 的新特性呢，看样子，我还真要好好研究研究 Oracle 10g 了。做 DBA 的，不进则退，不学习看样子是不行了。

转过念头又一想，这其实也没什么，毕竟 DBA 只是一个工作，而不是生活的全部。工作是为了更好地生活，而不是工作本身，因此也没必要对自己太苛刻了，上回吃饭时老方就说他坚决不会再学 Oracle 10g 了，准备就靠手头这些东西吃老本了，现在想想也对，都是奔四的人了，能把自己的经验传授给年轻人，就已经很不错了，再和年轻人去比赛新技术，怕是吃力不讨好了。

今日点评

使用临时表进行 SQL 优化是 DBA 经常使用的手段，就像老于帮助客户做的那样。开发人员喜欢在一个 SQL 中生成所有的数据，这也是 DBA 最头痛的。在做优化项目中老白也经常碰到一些 SQL，面有 N 个相类似的 SQL 做 `UNION` 或者 `UNION ALL`，这些 SQL 都是对相同的表按照不同的条件进行扫描。今天这个 SQL 就是一个有 10 个 `UNION` 操作的查询，只是其中的某个 `WHERE` 条件有略微的不同。这种情况下，这个 SQL 要对一个复杂的查询运行 10 次，然后才能得到结果。换一个思路，首先把这 10 个 `UNION` 操作所需要访问的数据查询出来，这部分数据可能只是原来数据的几十分之一甚至几百分之一，这部分数据先存在一张临时表中，然后对这张临时表进行查询，那么这个 SQL 的效率可能提高数倍甚至数十倍。

其实今天使用的优化方案还不是很彻底的，原本老白提出了一个更为彻底的优化方案，后来由于应用修改较大，并且通过调整后的应用性能已经能够满足客户要求了，因此就没有进一步优化。实际上，这个调整后的优化方案还是对临时表进行了 10 次扫描才产生了结果，其实如果把 SQL 修改为一个存储过程，那么只需要扫描一遍临时表，就可以生成这个 SQL 的结果了。比如下面的语句：

```
SELECT 'A',COUNT(*) FROM TMP WHERE STATUS='A'
UNION ALL
SELECT 'B',COUNT(*) FROM TMP WHERE STATUS='B'
UNION ALL
SELECT 'C',COUNT(*) FROM TMP WHERE STATUS='C'
UNION ALL
SELECT 'D',COUNT(*) FROM TMP WHERE STATUS='D';
```

如果上面的 SQL 需要对临时表扫描 4 次，那么可以改写为一个存储过程，具体如下：

```
DECLARE
  VA: INTEGER;
  VB: INTGER;
  VC: INTEGER;
  VD: INTEGER;
BEGIN
  VA:=0;VB:=0;VC:=0;VD:=0;
  FOR C IN (SELECT STATUS FROM TMP) LOOP
IF C.STATUS='A' THEN
```

```
    VA:=VA+1;
  ELSE
    IF C.STATUS='B' THEN
      VB:=VB+1;
    ELSE
      IF C.STATUS='C' THEN
         VC:=VC+1;
      ELSE
         IF C.STATUS='D' THEN
            VD:=VD+1;
         END IF;
      END IF;
    END IF;
  END IF;
     END LOOP;
     ......
  END;
```

优化小技巧　表访问的方式

今天的日记里提到了一个 SQL 优化的问题，其中涉及了使用 Hash 连接还是嵌套循环的问题，所以现在介绍一下和 SQL 执行计划有关的知识。在讨论表访问方式之前，我们先来熟悉一些和 SQL 执行计划有关的技术术语。

- Row Source：一个实现某种操作并且返回一个结果集的函数。
- Predicate：查询的 WHERE 条件。
- Tuples：返回的记录。
- Driving Table：驱动表，作为查询种子的行源（Row Source）。
- Probed Table：和驱动表连接的表。

在物理层，Oracle 读取数据的最小单位是块，Oracle 读取数据的最大限制取决于操作系统和 Oracle 对多块读 I/O 的限制。从物理上来说，一个 SQL 语句要读取某个记录，必须将该记录读取到 DB Cache 中，然后才能从中获取，这种访问我们一般称为物理读（READ）；如果这个数据已经存在于 DB Cache 中，那么前台进程可以直接从 DB Cache 中读取数据，这样的读取称为逻辑读（GET）；如果要读取的数据已经被修改，需要从 undo 表空间中读取前映像来获取一致性数据，那么会从 undo 表空间中取出前映像，和当前的数据块一起形成一个一致性读块（CR BLOCK），然后再从一致性读块中读取数据，这种访问方式称为一致性读（CR GET）。

从逻辑上讲，Oracle 通过 3 种途径读取数据：全表扫描（Full Table Scan，FTS）、索引扫描、通过 ROWID 直接访问。在阅读 SQL 执行计划时，可以通过 TABLE ACCESS 子句来查看 Oracle 访问某个表的方法。一般大型的表，如果出现 TABLE ACCESS FULL 的提示，就需要加以重视。一般情况下，对于大表的全表扫描，应该尽量避免。

了解访问的详细方式，有助于开发人员和 DBA 进行 SQL 优化时抓住重点，避免错误。因此，本节将详细介绍各种访问方式。

1. 全表扫描

在一次全表扫描执行中，整个表被扫描，直到高水位标志（High Water Mark，HWM）标示的位置。高水位标志指向该表包含数据的最后一个数据块的位置。如果你的表中有 1 000 万条记录，当你用 DELETE 语句删除了所有数据后，高水位标志仍然标示删除前最后一个块的位置，因此，如果这个时候对这张表进行了全表扫描，那么还是要读大量的数据块，而不是像大家想象的那样，不需要读任何数据块了。

如果我们进行一次 TRUNCATE TABLE 操作，那么高水位标志会被设置为表的第一个数据块，那样如果进行全表扫描，就不需要扫描那么多数据块了。由于全表扫描的这种特性，需要访问所有的高水位以下的数据块，所以需要尽量避免全表扫描。如果全表扫描是不可避免的，那么如果大量删除数据后，需要对表进行重组，否则全表扫描时还是会扫描所有的数据。

进行全表扫描时，是采用多块读的方式，多块读由 db_block_multiblock_read_count 参数控制。多块读的数据会被放置在数据块缓冲的 LRU 队列的尾部，这样这些数据会在尽可能短的时间内被换出。

2. 索引扫描

从索引中可以获得数据的 ROWID，通过 ROWID 可以直接定位到数据。ROWID 可以唯一性地定位到某一条记录的物理位置。看下面的例子：

```
SQL> explain plan for select empno,ename from emp where empno=10;
Query Plan
------------------------------------
SELECT STATEMENT [CHOOSE] Cost=1
TABLE ACCESS BY ROWID EMP [ANALYZED]
INDEX UNIQUE SCAN EMP_I1
```

如果所需要的所有信息可以直接从索引中获得，不需要访问表，那么执行计划是这样的：

```
SQL> explain plan for select empno from emp where empno=10;
Query Plan
------------------------------------
SELECT STATEMENT [CHOOSE] Cost=1
INDEX UNIQUE SCAN EMP_I1
```

索引访问也不一定全部是 UNIQUE SCAN，也可能会产生范围扫描，比如：

```
SQL> explain plan for
SELECT empno,ename
FROM emp
WHERE empno > 7876 order by empno;

Query Plan
--------------------------------------------------------------------------------
SELECT STATEMENT   [CHOOSE] Cost=1
TABLE ACCESS BY ROWID EMP [ANALYZED]
INDEX RANGE SCAN EMP_I1 [ANALYZED]
```

常见的索引访问模式有以下几种。

- index unique scan：一般是主键或者唯一性索引访问。
- index range scan：一般在查询条件里存在范围性条件。
- index full scan：按照索引的顺序进行全扫描。一般来说，如果发现全索引扫描的性能好于全表扫描，才会采用这种方式，并且扫描出来的数据是有顺序的。
- index fast full scan：扫描索引的所有块，返回的数据不是按照索引顺序的。
- index skip scan：索引跳跃扫描。如果 `WHERE` 条件的字段不是索引的键值的第一个键，在不支持 index skip scan 的系统中，这个索引是无法使用的；支持该访问方式的系统中，可以跳跃式扫描索引，完成访问。从 Oracle 9i 开始，Oracle 支持这种访问方式。

3. ROWID 访问

这是访问数据的最快方式，如果查询中存在 `WHERE ROWID=:X`，那么不需要通过任何索引，直接可以定位到这条记录的物理位置，访问该数据。

5月31日 电脑坏了

今天上午和孙主任确认了优化实施的时间，然后又分析了几个 SQL。通过前一段时间的工作，关键的 Top SQL 已经基本上找出了，剩下的就要看今天开始这几天半夜做的批处理里面有没有什么需要优化的了。之前我们已经把 STATSPACK 采样的周期缩短为 30 分钟，以便于更好地进行分析。本来想把 Snap Level 设置为 6，这样就能够采集到 SQL 的执行计划了，不过考虑到把 Snap Level 设置为 6 会加大 Snap 采样时的成本，CPU 就更吃不消了，最后只能放弃了。

老于这几天都在和老肖一起整理优化方案，由于事情比较多，抽烟也比平时少了不少。快中午时，我走过去拍拍正在对着显示器发呆的老于，找他一起出去抽根烟。大半个月过去了，这段时间大家绷得都很紧。不过下个月初的第一次优化才是最关键的，对于我们这个优化小组来说，起码也算上是一次期中考试，如果成绩不理想，那就麻烦了。优化项目的成败和客户的支持程度有很大的关系，如果客户丧失了对你的信任，那么你做的每个操作都可能面临相当大的压力。因为在系统优化中，任何一个操作都存在一定的风险。辽宁这个项目，孙主任的支持是一个很好的保障，这种情况下我们还敢稍微冒点险。在绝大多数优化项目中，甲方的人都希望能很快出成绩，调好了什么都好说，如果第一次实施后出现了副作用，就可能成为甲方的替罪羊，甚至可能马上被客户赶走。

在优化项目中出现一些反复也很正常，但是甲方可能并没有这样的准备。因为甲方的项目负责人直接对自己的领导负责，而领导是不知道数据库优化是怎么回事的。领导们一般会认为花钱请了专家来优化，就一定要看到效果，就像交换机容量不够了，花点钱扩容一下就没问题了。一旦出现优化后性能反而下降这种情况，领导的第一个反应是打板子，而这个板子一般不会直接打到乙方身上，会打到甲方的项目负责人身上。如果甲方的项目负责人挨了领导的批评，甚至被扣了绩效，那么这股邪火可能会转嫁到乙方身上。我就曾经做过一个优化项目，进场后不久，甲方的项目主管就很愤怒地找到我，问我为什么做了优化反而性能下降了。我当时很诧异，因为项目刚刚开始，这些天我们一直在进行数据采集，根本没有做任何优化动作。我就告诉那个领导，我们这几天只是在分析系统，采集数据，并没有做任何操作。他说为什么我们来之后系统性能会下降呢，是不是你们采集数据影响了系统性能。我告诉他我们采集数据基本上就是执行几个简单的 SQL，不会对系统有多大的影响，而且我们做采集时会尽量避开业务高峰时段。这几天系统性能下降是由于最近的业务量比平时大了差不多 30%。他听了以后如释重负，说只要不是我们搞出的事情，他就没责任了。他马上拿起电话，和领导诉了半天苦。然后又和我说，为什么不早点告诉

他这几天系统负载比平时大，害的他今天上午开局务会时被业务部门和领导猛批了一顿。最后很严厉地告诉我，以后碰到这种情况一定要第一时间通知他。

中午吃晚饭后，我和老于一起在门口抽了根烟，才进办公室。走到电脑面前，发现电脑的屏幕是黑的，本来以为是电脑待机了。按了一下电源，发现没有任何反应。拔掉电池，重新插上，再开机，还是没反应，看样子是出问题了。马上让老于上网搜了一下，找到了一家三好街的 HP 维修站，马上打车过去。那哥们儿打开电脑看了看，说 CPU 坏了，还好在保修期内，可以免费更换主板，不过主板要从北京发货，所以要下周一才能修好。没办法，拆下硬盘，把电脑交给维修站。从维修站出来后，马上联系小齐，看看能不能借一台笔记本。在等小齐回电话时，我在旁边的配套市场里买了一个移动硬盘盒子，如果借到电脑，就可以把盘里的数据先拷上去。

半个小时后，小齐那边有回复了，她在沈阳 HP 公司借到一台笔记本电脑，我可以直接去拿。沈阳 HP 公司在青年大街上，离三好街很近。经过一个小时的努力，填写了无数表格，这台电脑终于归我了。打开电脑，输入 HP 公司登记的用户名和密码，居然说密码错误。又经过一阵子折腾，终于找到了电脑原来主人的手机。我拨通了电话，电话那头传来一个熟悉的声音，原来这台电脑原来是明喻的。我们也算老相识了，2004 年做海尔项目时，明喻是海尔项目的项目经理。明喻已经离开 HP 公司，那台电脑也已经交了快 1 年了。他把常用的几个密码通过短信发给我，试到第三个时，居然成功了。开机时感觉这台电脑比我那台要慢得多，开机以后确认了一下，原来是一台奔腾 II 的电脑，怪不得这么慢了，不过还好，内存有 512MB，可以凑合着用。电脑虽然拿回来了，不过安装软件又是一件麻烦事。Oracle 9i 客户端是必须安装一个的，包括全套的 OEM 客户端，这是我优化项目时必须使用的工具。另外，Toad 也是我最喜欢的工具，一直用不惯 8.0 版本，所以我一直还在使用 Toad 7.6。Toad 有几个很好用的小工具，一直是我做优化分析时必用的，比如说 Toad 的健康检查、SGA Trace 和 SCHEMA BROWSER。Toad 的 SCHEMA BROWSER 可以查看某个对象的大小，我经常用来对表和索引进行分析。SGA Trace 功能也十分有用，可以从 SGA 中抓取 SQL 语句，并按照某个指标值进行排序（比如缓冲区获取），在使用 OEM Top SQL 之前，我一直在使用 Toad 的 SQL Trace 来分析 Top SQL。

除了 Oracle 数据库工具外，Telnet 工具也是必需的。一般来说，我会使用 NetTerm 和 Xmanager。平时我比较喜欢用 NetTerm，我这个人不太喜欢尝试新鲜事物，用工具本着够用就行的原则。NetTerm 对我来说基本够用了。我使用 Net Term 也就是远程登录到服务器上，同时通过会话日志记录我的整个操作过程，其他功能基本上很少使用。Xmanager 是在需要使用 X-Windows 时的首选，不过在平时做项目时，Xmanager 使用的机会不多，大多数是在做数据库安装时使用。

除了上面的工具外，UltraEdit 也是我喜欢的工具。这个软件很小巧，功能也不错。回到办公室后，首先安装了一套 Oracle 9i 的数据库。原本准备装一个客户端的，后来想想有时候可能还要做一些试验，所以装个 RDBMS Server 也许能有点用。借来的这台机器确实比较慢，装个 Oracle 9i 都花了将近两个小时。

下午由于出了这档子事情，感觉时间过得很快，转眼就到下班时间了。客户那边做账务处理

的人也陆续到了这个平时很少来的办公室，准备晚上 8：00 开始账务处理。临走前，为了更好地监控系统在半夜账务处理时的状态，我把 OEM Performance Manager 也启动了，开启了其中的健康性图表、平均事务响应时间、文件成本、闩锁等几个重要的视图，设置采样时间为 1.5 分钟，这样白天就可以通过对 EM 的回放来分析数据了。

今日点评

DBA 的电脑上需要安装什么软件？经常有人问老白，其实每个 DBA 都有自己喜欢使用的软件。对于使用什么软件，用得习惯、用得熟练就好。因为工具只是帮助 DBA 思考，在 DBA 进行分析时提供辅助作用。因此，每个 DBA 使用的软件一定要和自己的分析习惯相吻合。对于初级 DBA 来说，Quest 公司的 Spotlight 这样的工具十分有效，可以初步分析数据库的总体情况。而对于资深 DBA 来说，Spotlight 工具能够提供的帮助就比较少了。

今天老白找来了一台替代的新电脑，虽然比较烂，不过也聊胜于无了。拿到电脑后，老白装了一系列软件。老白的电脑上都有些什么工具软件也是很多朋友经常问的。实际上，老白的电脑上的工具软件很少，具体如下所示。

首先老白的电脑上肯定会装有数据库，Oracle 9.2.0.8 和 Oracle 10.2.0.3 的数据库各有一个。在 Oracle 9.2.0.8 的数据库里还配置了一个 OMS 的服务。

Oracle 9i 的数据库里当然安装了老白的最爱 OEM。OEM 工具中老白经常使用的是诊断包和优化包。

Toad 是老白十分喜欢使用的工具，不过老白的电脑里安装的还是 Toad 7.6。对于工具，只要够用就行，并不是越新越好。

Spotlight 是个不错的工具，当然应该安装。不过使用这个软件的机会不多，因为 Spotlight 要在用户的数据库中创建很多对象才能使用好分析功能，而对于老白服务的绝大多数客户，都不允许随便创建表，所以也只能忍痛割爱了。

实际上说，老白的电脑中和 Oracle 相关的工具只有上面这几个了。老白陆续使用过其他工具，但感觉和他分析问题的思路不一致，因此也就没有继续使用。除了 Oracle 工具外，其他一些系统工具也是十分重要的。

NetTerm 是远程登录工具，而且有免费版的。NetTerm 的兼容性很不错，主流的操作系统下表现都不错，而且这个软件可以免安装，所以每次老白换电脑时只要把安装目录复制过去就行了，里面的服务器定义不会丢失。NetTerm 的会话日志功能也十分好，每次老白连到客户系统上时都会开启会话日志，这样就可以把每次操作的情况记录下来。老白写这本书时，很多细节都是借助会话日志才回忆起来的。老白不喜欢使用支持多页的 Telnet 工具，这不是说多页支持不好，而是在老白的 DBA 生涯中越做越谨慎了。五六年前老白还是很喜欢一次性开多个窗口来进行操作的。而随着 DBA 工作经历的增加，现在老白很怕在同时开多个窗口进行操作。如果必须开多个窗口，那么绝对要保证，多个窗口都是连到同一台服务器的同一个 Unix 账号的。这是为什么呢？作为

一名 DBA，没有过误操作经历的人可能很少，有时候误操作只能让你出身汗，而很多时候，误操作可以让你万劫不复。大概七八年前吧，老白碰到过一件事，当时在一个客户那里做服务，和开发商的人坐在一个办公室里。当时开发商有个哥们刚刚坐火车回到客户那里，正好碰到两件事情：一件是生产库上有个进程挂住了；另外一件是测试小组要开始新一轮测试，需要清理一下测试机的数据库。因此，那个哥们开了两个窗口，分别进行操作，由于当时可能刚刚坐了一夜火车，在这两个系统上来回倒了几次后做了一个误操作，删除了一张重要的生产表的数据。从那次以后，老白在客户现场尽可能避免同时连到多个系统上，以避免不必要的误操作。

ssh (secure shell)：连接 ssh 的服务器的工具，不需要老白做过多解释了吧。如果你觉得这个工具太土，那么也可以选择 SecureCRT。

Xmanager：这个恐怕不用说了，不过使用的机会并不多，主要是在做数据库安装和升级时使用。

UltraEdit 也是老白重要的武器，其最大优点是打开的文件变化时能够捕捉到变化，并重新更新。因此，老白经常使用 UltraEdit 来读取 NetTerm 的会话日志文件。

Cygwin：一个可以在 Windows 下模拟 Linux 的软件。在这个软件里，可以在 Windows 下使用 Linux 的命令和程序，比如 `dd`、`awk`、`gc++`等。直接在 Windows 下用 `awk` 调用 ass 分析系统状态转储是十分有用的。有时候老白需要写个简单的 C 程序，也可以用 Cygwin 来调试。

Wincvs：它是著名的文档版本管理软件 CVS 的 Windows 版本，用来管理文档。

Firefox+scrapbook：老白的知识库收集工具。在网上看到有什么好的文档，立即拉到知识库里。

优化小技巧　通过 OEM 录像采集数据

Oracle 9i 没有 ASH 的功能，因此不能进行事后的事件追踪。不过依赖于监控软件，可以实现事后回溯。事后回溯对于分析问题十分关键。在 Oracle 9i 上，通过 OEM 的图表录像可以实现这个功能。对于重点时间段采用 OEM 的图表录像功能，可以基本上记录下系统状态的主要方面。今后通过图表的回放功能，就可以查看某个时段的系统状态。

一般来说，如果要做性能调优，下面几个图表是十分重要的：

- 系统健康性图表（这是一个总图，可以看到系统的整体情况）；
- 系统响应时间图表；
- I/O 成本图表；
- 闩锁汇总。

使用图表录像功能，建议使用 OMS 服务器。如果不使用 OMS 服务器，那么程序一旦退出，录像就无法再次观看了。因为 Oracle 9i 的 OEM 客户端是用 Java 写的，很容易死掉，所以不连接 OMS 服务器，可能会导致录像无法观看。

Oracle 工具的功能确实很不错，不过用户界面不友好是顽疾。Oracle 9i OEM 的回看功能只有原速播放和快进两种模式，不能拖拉也不能设定更快速度的快进。如果录像很长的话，回放将

会是十分痛苦的事情，因此建议录像的长度不要太长。我曾经尝试过在一个 10 小时的录像中找 20 分钟的数据，快进了 1.5 小时才找到那段数据。看完一遍后如果还想再看一遍，又需要再快进 1.5 小时，而不能后退。由于 OEM 存在的问题，在回放过程中，如果看到什么有用的线索，一定要马上通过报告功能生成一份报告，这样就不需要多次回放来查看某个数据了。

优化小技巧　调整游标相关参数

SQL 语句分析分为软分析和硬分析两种。减少软分析和硬分析，特别是减少硬分析，对于降低 CPU 的使用率有着十分关键的作用。

一般来说，SQL 分析和应用软件有着十分紧密的联系。优化应用软件是彻底解决 SQL 分析问题的最佳方法。但是在数据库优化时，往往很难对应用软件进行彻底优化，因此通过调整系统参数来优化 SQL 分析是 DBA 经常面临的问题。

SQL 执行时，如果某个语句已经是会话缓冲的（在会话的游标缓冲区中存在），那么这个 SQL 就不需要进行分析，可以直接执行，因此保证 SQL 能够在缓冲区中长时间存在可以减少 SQL 分析的发生。有两个参数可以控制 SQL 在会话缓冲池中的时间长短：open_cursors 和 session_cached_cursors。open_cursors 参数控制每个会话可以打开的游标的最大数量；session_cached_cursors 控制每个会话可以缓冲的游标的数量。在 Oracle 9i R2 之前，往往通过 open_cursors 来控制每个会话可以打开的游标的数量。从 Oracle 9i R2 开始（特别是从 Oracle 9.2.0.5 开始），通过 session_cached_cursors 的会话缓冲游标机制可以使会话尽可能多地使用被缓冲的游标，从而避免 SQL 被重新分析，这样可以大大降低软、硬分析的数量。

在设置这两个参数时，应该注意 session_cached_cursors 的值不能超过 open_cursors 的值。在 Oracle 9i 中，参数 session_cached_cursors 的默认值为 0，将其设置为任何大于零的值将启动 Oracle 的这个功能（Oracle 10g 默认开启了这个功能）。在设置 session_cached_cursors 参数之前，首先要确定共享池的大小是否足以支持缓冲这些 SQL。因为 session_cached_cursors 是针对每个 SESSION 的，所以对于拥有几百甚至上千个 SESSION 的 OLTP 系统，设置 session_cached_cursors 时要十分注意。设置大的 session_cached_cursors 参数需要比较大的共享池来支持，如果调整了这个参数后出现共享池空间不足的情况，调整共享池的大小或者减少 session_cached_cursors 参数就是 DBA 应该进行的操作。通过下列语句，可以检查当前参数 session_cached_cursors 和 open_cursors 的情况，根据查询结果，可以确定如何调整这两个参数：

```
SELECT
  'session_cached_cursors'  parameter,
  lpad(value, 5)  value,
  decode(value, 0, '  n/a', to_char(100 * used / value, '990') || '%')  usage
FROM
  ( SELECT
      max(s.value)  used
    FROM
```

```
        v$statname  n,
        v$sesstat  s
      WHERE
        n.name = 'session cursor cache count' and
        s.statistic# = n.statistic#
    ),
    ( SELECT
        value
      FROM
        v$parameter
      WHERE
        name = 'session_cached_cursors'
    )
  UNION ALL
  SELECT
    'open_cursors',
    lpad(value, 5),
    to_char(100 * used / value,  '990') || '%'
  FROM
    ( SELECT
        max(sum(s.value))  used
      FROM
        v$statname  n,
        v$sesstat  s
      WHERE
        n.name in ('opened cursors current', 'session cursor cache count') and
        s.statistic# = n.statistic#
      GROUP BY
        s.sid
    ),
    ( SELECT
        value
      FROM
        v$parameter
      WHERE
        name = 'open_cursors'
    )
  /
```

在上述查询中，如果返回 session_cached_cursors 缓冲区的使用率是 100%，那么说明 session_cached_cursors 参数还不够大。如果共享池的大小足够，可以调整该参数，直到使用率低于 100%为止。

对于没有使用绑定变量的系统，如果 cursor_sharing 设置为 exact，那么设置 session_cached_cursors 时要十分注意。由于应用原因，游标的重用率十分低。如果设置过高的 session_cached_cursors，会导致共享池空间被大量占用，在系统负载较高时会出现共享池的性能问题。

另外一个十分重要的问题是，SQL 分析带来的性能问题往往会被夸大。如果某个系统中的 SQL 分析带来的开销很小，比如只有 1%～5%，那么需要充分考虑我们是否很必要来进行优化。因为调整游标相关的参数，可能带来共享池使用率的增长，有些错误的调整甚至可能增加宕机的

风险。我曾经碰到过一个客户，他们做了优化后，系统十分容易宕机，宕机的原因是 ORA-4031。我给他们分析时发现，他们上次优化时加大了 session_cached_cursors，同时将 CURSOR_SPACE_FOR_TIME 设置为 true。这是一个风险十分大的调整。就是这个调整导致了频繁的宕机，将 CURSOR_SPACE_FOR_TIME 修改为 false 后，这个问题就解决了。我问他们为什么做这个调整，他们的回答让我啼笑皆非："经过一年的优化，系统已经很不错了，只有 SQL 软分析比例还是比较高，这个月没别的地方可以优化的了，就只能做这个优化了。"为了优化而优化，这种教条主义往往是会害死人的。

6月1日 ORA-4030 和内存泄漏

昨天晚上走的时候把 OEM 的屏幕录像开了，所以今天一大早过来就看了看昨天晚上录制的数据，快进了半天也没看到什么特殊的东西，干脆就不看了。打电话问了几个昨天晚上做账务处理的人，大家都说比较正常，除了速度比上个月更慢了一些以外，没有性能下降特别多的模块。账务处理的速度每个月都是线性下降的，所以大家都觉得很正常，并没有什么特别的地方。既然大家都没觉得什么，昨天晚上录制的东西也就懒得去看了。不过看录像时突然发现，这台安装 OMS 服务器的微机速度比我那台借来的老掉牙的笔记本快多了，干脆用这台电脑工作吧，用那台电脑起码降低 1/3 的工作效率。

优化方案基本上都确定了，剩下的事情老于他们在做细化，细化工作包括整理优化实施方案和相关的脚本。SQL 优化的工作有老熊帮忙，我也轻松了许多。前面几批提交的三四十个 SQL 优化方案也分析得差不多了，大部分都已经由张工转发给了开发商。由于昨天电脑故障，今天我主要的工作还是在现有的电脑上安装相关的工具，复制相关的文件。

这几天系统虽然负载很重，不过这种情况已经持续了快一年了，所有的人也都没有感觉什么不适应的地方。中午快吃午饭时，张工突然打电话过来，说他们那边有一个数据库出现了 ORA-4030 错误。由于那套系统并不在我们合同范围之内，所以张工很委婉地问我有没有时间，帮他们看看。

今天反正事情也不多，我很爽快地答应了下来。使用张工给我的账号和密码登录到了系统里，发现报错的信息是：

```
ORA-04030: out of process memory when trying to allocate 2097192 bytes (joxcx
callheap,ioc_allocate ufree)
Current SQL statement for this session:
call CRM_OPER.oper(:1,:2)
----- PL/SQL Call Stack -----
  object       line  object
  handle     number  name
c000000100da8108         0  package SYS.XMLNODECOVER
c000000100dada80       381  package body SYS.XMLDOM
c000000103a1d178        77  package body CRM.CRM_OPER
```

看到这个错误信息，首先我查看了一下 ulimit，发现 ulimit 设置很正常，全部是 unlimited，没有可能导致故障的设置。检查物理内存的使用情况，发现物理内存还有一些空闲，虽然不是很多，不过也还是有 300 多兆空闲。交换区的使用率相当高，高达 98%，从这一点上可

以看出，不久前系统曾经出现过十分严重的换页现象。从调用栈中可以看出，这是 CRM.CRM_OPER 在调用 xmldom 包时报错。DOM 操作是十分消耗内存的，这回报错的是一次申请 2MB 的内存空间。看到 DOM 操作，我的第一个反应就是，是不是 dom 对象使用后没有释放？看样子必须分析 CRM_OPER 的代码。用 Toad 找到了这个存储过程，幸运的是这个存储过程并没有进行 WRAP。CRM_OPER 代码并不长，主要是生成一个 XML 数据结构，总共也只有 50 多行。很快下面这段代码引起了我的注意：

```
    parser :=   xmlparser.newParser;
    xmlparser.parseBuffer(parser,operInfo);
    doc := xmlparser.getDocument(parser);
xmlparser.freeParser(parser);
```

虽然 xmlparser 被释放了，但 documentation 对象 doc 并没有释放。doc 是一个 dom 对象，如果不释放，dom 对象所占的空间将会随着调用的增加而叠加，最终导致物理内存被耗尽。为了证实这个猜测，首先编写一个测试程序，具体如下：

```
create or replace procedure oper is
    parser                xmlparser.Parser;
    doc                   sys.xmldom.DOMDocument;
    operInfo    varchar2(2000);
    i           integer;
  begin
    operInfo:='<?xml version="1.0" standalone="yes" ?>';
    operInfo:=operInfo||'<QUERY_LIST>';
    operInfo:=operInfo||'<LIB NAME="areasQueries" VER="2.2.0.7.0" REP_VER="0.0"
MIN_INST_VER="2.1.0.4.1" INV_LOC="Queries21/areasQueries/2.2.0.7.0/areasQueries.jar"/>';
    operInfo:=operInfo||'</QUERY_LIST>';
    i:=0;
      parser :=   xmlparser.newParser;
      xmlparser.parseBuffer(parser,operInfo);
      doc := xmlparser.getDocument(parser);
      xmlparser.freeParser(parser);
--      xmldom.freeDocument(doc);
      i:=i+1;
  end;
/
```

分别进行两次测试，第一次不执行 xmldom.freeDocument(doc)，第二次执行 xmldoc.freeDocument(doc)。通过下面的方法调用：

```
declare
i integer;
begin
  i:=0;
  loop
      oper;
      i:=i+1;
      exit when i>100;
  end loop;
end;
/
```

每调用一次，通过下列语句检查 PGA 的情况：

```
select value, n.name|| '('||s.statistic#||')'
   from v$sesstat s , v$statname n
   where s.statistic# = n.statistic#
   and n.name like '%ga memory%'
   and sid= <sid of problem session>;
```

首先进行一次调用 `freeDocument` 的测试，查看 UGA 和 PGA 的使用情况。调用前的结果是：

```
-------------------------------------
    142476 session uga memory(15)
    207940 session uga memory max(16)
    710024 session pga memory(20)
   1037704 session pga memory max(21)
```

第一次执行后的结果为：

```
-------------------------------------
    207940 session uga memory(15)
    273404 session uga memory max(16)
   8165056 session pga memory(20)
   8165056 session pga memory max(21)
```

第二次执行后的结果为：

```
-------------------------------------
    207940 session uga memory(15)
    273404 session uga memory max(16)
   7882672 session pga memory(20)
  10734088 session pga memory max(21)
```

可以看出，随着调用次数的增加，PGA 和 UGA 并没有明显增长。下面修改一下测试程序，将 `freeDocument` 的部分去掉。没执行时 PGA 的情况为：

```
-------------------------------------
     77012 session uga memory(15)
    142476 session uga memory max(16)
    776488 session pga memory(20)
    842024 session pga memory max(21)
```

第一次执行后的结果为：

```
-------------------------------------
    142476 session uga memory(15)
    207940 session uga memory max(16)
   5188024 session pga memory(20)
   5253560 session pga memory max(21)
```

第二次执行后的结果为：

```
-------------------------------------
    142476 session uga memory(15)
    207940 session uga memory max(16)
   5126608 session pga memory(20)
   5253560 session pga memory max(21)
```

通过 Windows 任务管理器观察 Oracle 进程的物理内存情况，可以得出类似的结果。如果释放 dom 对象，PGA 内存不会持续增长；如果不释放 dom 对象，PGA 内存会越占越多。通过 Windows 任务管理器可以看出，执行 200 次存储过程，Oracle 进程的内存使用增加了 200MB。从上述测试中可以得出，由于没有释放 document 对象，导致了内存泄漏。我把这个结果告诉了张工，他和开发人员确认了一下，确认这个模块是昨天下午才上线的一个应用调用的，程序员也确认不释放 document 对象是错误的。

处理完这个小故障，已经是中午 12：30 了。办公室里除了老于其他人都去吃饭了。我急忙锁了屏幕，和老于下楼去吃饭。中午吃饭时，老于一直比较沉默，我问他是不是有什么心事。他说有点想家，老于家就是锦州的，原计划这周末回家休息一下，由于 6 日要做实施，这个计划只能泡汤了。做 DBA 的，很多像老于那样，一年到头没有多长时间能够和家里人团聚。有时想想，对家人亏欠确实很多。

今日点评

关于 ORA-4030 的分析，Metalink 上有几份文档很具有参考价值。

- Note:116076.1 Tackling ORA-4030 on Windows NT。
- Note:233869.1 Diagnosing and Resolving ORA-4030 Errors。
- Note:199746.1 How to Resolve ORA-4030 Errors on UNIX。
- Note:395800.1 Things to Know about Error ORA-4030 in AIX Environments。
- Note:373602.1 How to Avoid ORA-04030/ORA-12500 in 32 Bit Windows Environment。
- Note:46001.1 Oracle Database and the Windows NT Memory Architecture, Technical Bulletin。
- Note:399497.1 FAQ: ORA-4030。
- Note:278310.1 ORA-4030 with Parallel Queries。

优化小技巧　如何分析 ORA-4030

ORA-4030 一般来说是进程从操作系统申请内存时操作系统无法分配内存所致。一般来说，ORA-4030 可能由几方面原因引起：操作系统无法从物理内存和交换区（交换区）分配内存给进程，或者进程受到了操作系统的某个限制条件的约束无法分配更多内存，还有就是碰到了某个 Oracle 的 bug。

遇到 ORA-4030 时，首先我们要检查的是操作系统的物理内存和交换区是否满了。如果物理内存所剩无几，并且交换区接近或者达到 100%，那么就是真正的内存不足了。一般情况下，这种现象出现的较少，大多数 ORA-4030 是由进程或者操作系统参数限制引起。

如果发现物理内存还有很多剩余，那么下一步需要检查的是 Oracle 账户的 ulimit 参数。比如：

```
SUN OS:
/tmp/oralog>>> ulimit -a
time(seconds)        unlimited
```

```
file(blocks)          unlimited
data(kbytes)          unlimited
stack(kbytes)         8192
coredump(blocks)      unlimited
nofiles(descriptors) 256
vmemory(kbytes)       unlimited
HP-UX:
$[/oracle]ulimit -a
time(seconds)         unlimited
file(blocks)          unlimited
data(kbytes)          1048576
stack(kbytes)         131072
memory(kbytes)        unlimited
coredump(blocks)      4194303
nofiles(descriptors) 2048

AIX:
[/home/oracle9i]#ulimit -a
time(seconds)         unlimited
file(blocks)          unlimited
data(kbytes)          unlimited
stack(kbytes)         65536
memory(kbytes)        unlimited
coredump(blocks)      2097151
nofiles(descriptors) unlimited
```

其中的 data、stack、memory 等参数过小都可能引起 ORA-4030。一般来说，建议将这些参数设置为-1(unlimited)。如果 ulimit 参数没有问题，那么也可能是其他操作系统内核参数引起的。我们可以使用下面的程序去测试一下一个进程能够分配到的物理内存的大小：

```
/* mem.c */
#include <stdio.h>
#include <errno.h>
main()
{
 long int i;
 i = 1;
 for(;;)
 {
  if (!malloc(1048576))
  {
   printf("malloc failed with error: %u at iteration %u\n", errno,i);
   exit(-1);
  }
  i++;
  printf("allocated %u M of memory\n", i);
  if ((i*1048576) >= 500000000) {
    printf ("Leaving...\n");
    exit(0);
   }
  }
}
```

上面的程序来自于 Metalink 文档 395800.1。一般来说，操作系统下都有 C 编译器，直接编辑 mem.c，然后编译执行一下就可以了。

对于 32 位操作系统，由于操作系统寻址的关系，物理内存是受到一定限制的，比如 Windows 操作系统每个进程能够使用的物理内存寻址空间是 2GB（32 位的最高限制）。而在 Windows 下，Oracle 是使用多线程模式的，所有的 Oracle 会话使用相同的进程空间，因此在 Windows 下，SGA 和会话使用的总内存不能超过 2GB。

另外，还有很多 bug 会导致 ORA-4030。Metalink 文档 399497.1 里面有很详细的介绍。这里篇幅有限，就不做详细介绍了。

6月2日 优化方案

今天是原定的优化方案的提交日期。早上一到办公室，老于就打电话给张工，希望今天能够和开发商一起讨论一下优化方案。张工和开发商联系后反馈回来的结果是，开发商这边没有专职的 DBA，因此无法对优化方案进行评审。这种情况下，方案评审就只能是我们和甲方一起做了。这样就变成了我们向张工他们介绍 6 日要做的事情，也就没必要到总部去搞正式会议了。方案评审就约在我们办公的地方，时间是下午 2：30。

原本以为在我们办公室做的方案研讨实际上是走过场，但下午的方案讨论会开得并不轻松。我们准备了详细的优化方案，每个步骤都写得十分清晰。优化实施从晚上 10：00 开始，预计到凌晨 4：00 结束，耗时 6 个小时。如果出现问题，回退点设置为凌晨 5：00。不过客户对每个步骤都提出了疑问。我们只是设定了一个操作窗口，但是并没有将每一个操作步骤的耗时明确列出。比如我们的操作中涉及对几张表做分区，对这几张表做分区需要多长时间？准备串行操作还是并行操作？

说实在的，由于没有进行严格的测试，我对一些时间点也只能进行估算，无法得到一个很明确的时间。这个问题讨论了很久，最后我建议今天晚上做一个类似的模拟测试，把时间估算准确，然后再修改一下这个方案。张工同意了这个方案，并且帮我们联系了现场运维人员，临时加了一个 40GB 的表空间给我们使用。方案的其他部分没有什么疑问，很快就得到了确认。对于保障和回退方面的问题，由于整个实施过程的基调就是可迅速回退，完善的操作性极强的回退方案得到了客户的认可。

由于晚上要做测试，所以大家都没有回去，晚饭是在对面的“北方大厦”吃的，也就是我们中午常去吃自助餐的那个地方。晚餐价格和中午一样，不过伙食比较差，好像是把中午的残羹冷炙加热了一下。吃饭的时候，大家都对 6 月 6 日的优化表示了强烈的信心。通过这半个多月的工作，特别是这几天对优化方案的分析，对目前系统的情况已经了如指掌了，这次优化能够提升的性能也不是我们以前在会议上承诺的 20%左右，按照保守估计，这次优化后系统性能应该能够提升 30%以上。实际上，目前除了 SQL 优化无法控制外，其他方面一切尽在掌握中。如果开发商能够配合得好一点，优化完成时系统性能翻番是不成问题的。不过从目前客户的情况来看，他们要求的不仅仅是数据上 30%的提升，而是彻底解决目前的问题，使系统能在两年内稳定运行，直到退网。

晚上的测试是要验证一下表分区操作所需要的时间是不是和我们预期的一致。我们找了一张

10GB 左右的表，首先在一个测试账号下面创建了另一张测试表，然后从这张表中复制数据。为了防止 undo 表空间不足，我们采用了串行方式，每次复制一个分区的数据，数据全部复制完毕后再创建索引。这张 10GB 的表总计耗时 70 分钟才重建完成。按照这种速度，是无法在 5 个小时内完成所有分区表的操作的，因此我们需要使用并行方式。并行方式的测试结果还不错，整个操作在 50 分钟内完成了。为了提高创建索引的性能，我们把 PGA_AGGREGATE_TARGET 加大了一倍。通过评估，即使使用并行操作，6 小时的维护窗口也存在较大的风险。

这样的话我们的优化实施无法在一个晚上完成，分成两个晚上。连续两个晚上对同一套数据库实施停机操作，客户方面也无法承受。采用两张表并行操作，从测试结果来看，效果不佳。上述的测试和我们预估存在的差异较大，主要是客户的硬件确实比较差，能挖的潜力都已经被我们挖得差不多了。测试陷入了困境，没想到大家认为天衣无缝的优化方案，居然存在这么大的一个问题。和客户一起讨论了一下，张工的意见是如果不行就分为两天实施，他去和业务部门协调。不过哪怕客户同意两天实施一套系统，那么整个实施过程需要整整一个星期，时间上也有点无法接受。如果一个晚上搞两套系统，又怕影响面太大，不好控制，一旦出现问题，将很难协调。大家都陷入了两难的境地。

通过激烈的讨论，最后终于拿出了一个方案。这个方案只有两点：一是向局方申请的停机窗口从 8：00 开始，到第二天凌晨 5：00 结束，这样我们就有 9 个小时的维护窗口；二是除了要分区的几张表，表分析操作提前到 6 日下午开始，争取在 8：00 实施之前全部做完。

按照这个思路重新排了一下实施方案。如果一切顺利的话，我们将可以在凌晨 2：00 之前完成所有的优化工作。我们的停机窗口里有 3 个小时的机动时间，这样就有了足够的保障。而且如果到 5：00 优化无法完全完成的话，离开始营业还有 3 个半小时。我们有充分的时间去执行回退预案。

最终的方案得到了张工的认可。明天我们把方案修改完毕，然后提交给张工，这个工作就算完成了，下一步就看 6 月 6 日了。

今日点评

今天老白他们在做优化实施方案。一个好的实施方案是十分重要的，老白参加过无数次实施，也有很多经验和教训。每次失败的实施都和优化实施方案准备不足有关，哪怕是很简单的实施操作，如果在前期方案准备时不充分，也可能出问题。对于十分重要的实施操作，在有条件的情况下，尽可能找个测试环境进行一下测试，找到方案中可能存在的不足。

如果没有条件搭建测试环境，那么实施方案的评审工作更为重要，最好能够让开发人员也参与评审工作。因为 DBA 可能对应用的情况不是十分了解，开发人员的参与可以从另外一个角度提出一些建议。今天老白他们没有得到开发人员的支持，这是十分遗憾的，因此老白和老于决定做一次实际的测试。虽然没有测试环境，但是晚上在生产环境做一个测试也是无奈之举。幸运的是，从测试中发现了很多问题，最终调整了优化实施方案。如果到了 6 月 6 日再出现这个问题，

那么老白他们只能进行回退了。在优化项目中，回退是很难接受的结果，可能会导致客户对优化团队失去信心，要尽量避免。

另外，关于停机窗口的选择，也是一个十分重要的问题。经验不足的 DBA 往往在设计停机窗口时过于轻率，由于没有对实施计划进行很好的设计，往往对可能出现的问题估计不足，所以在实际实施过程中，往往由于某些突发事件而手忙脚乱。有经验的 DBA 往往会给自己留有足够的余量，尽可能让自己获得更多的资源支持。记得有一次做一个 RAC 添加节点的项目，按照设计的预案是不需要停数据库的，不过为了保险起见，我还是向客户申请了 4 个小时的停机窗口。刚刚开始时进展都很顺利，甚至客户都开始嘲笑我过于神经过敏了。这个时候突然出了一个 ORA-600，必须停数据库打补丁。如果当时我们没有申请停机窗口，那将是一次事故，而由于提前做了申请，事故就被避免了。

优化小技巧　一个提供参考的优化方案

有些 DBA 觉得优化方案是十分神秘的东西，总觉得优化方案里应该包含一些不为人知的高深的东西。而事实上，一个能解决问题的优化方案才是最好的优化方案，而能解决问题的优化方案往往都是朴实无华的，其中大多数技术都是我们平时耳熟能详的东西。比如，我们针对这个项目提出的优化方案包括操作系统优化和数据库优化两个部分。

其中操作系统优化包括：

- 将 maxclient%从 80 调整为 20；
- 将 maxperm%从 80 调整为 20；
- 将 minperm%调整为 10。

数据库参数调整包括：

- 设置 lock_sga=true；
- 加大共享池；
- 加大 pga_aggregate_target；
- 加大 DB Cache；
- 设置 session_cached_cursors；
- 设置 cursor_sharing=similar；
- 增加 db_writer_processes；
- 减少 large_pool_size；
- 减少 java_pool_size；
- 加大 log_buffer；

重做日志文件调整包括：

- 将重做日志文件大小加大为 1GB；
- 增加重做日志组的数量；

表空间调整包括：

- 临时表空间增加 4 个大小为 2GB 的临时文件；

其他调整包括：

- 将部分 PL/SQL 对象缓存到共享池；
- 分析所有的表和索引。

优化小技巧　游标的共享

在优化方案里提到了两个参数的调整，就是 cursor_sharing 和 session_cached_cursors，这两个参数都是和游标共享有关的。Oracle 的目的就是让一个游标可以分析一次，执行多次。因此将游标的可共享部分存放在库缓冲（library cache）里。

Oracle 判断游标共享的第一个原则是，可共享的游标的 SQL 文本必须完全相同。游标在执行前，首先对其文本计算散列值，通过散列值找到相同的游标。如果找到了相同的游标，而且该游标的所有对象（包括 subheap）都是有效的，那么这个游标在执行时，可以使用共享的 SQL。如果某些对象已经被换出，那么这个游标就需要进行软分析，将丢失的部分补充完整才能执行。

如果 SQL 文本完全相同，是不是就能够共享呢？实际上还是不行的，比如有 SCOTT 和 TIGER 这两个用户，下面都有 test 表，如果在这两个用户下都执行 SELECT 1 FROM tt WHERE rownum <2，而这两个 SQL 访问的表是不同的，那么这两个 SQL 是不应该共享的。Oracle 在这种情况下是怎么处理的呢？首先，由于 SQL 文本完全相同，所以这两个 SQL 具有相同的 SQL_ID 和散列值，被认为是相同的 SQL，在 v$sqlarea 下可以看到如下结果：

```
SQL_ID          ADDRESS   SQL_TEXT                            VERSION_COUNT
cpjnybv7021rv    1F7059E0    SELECT 1 FROM tt WHERE rownum<2      2
```

在 v$sql 中可以看到如下结果：

```
SQL_ID          ADDRESS        SQL_TEXT
------------------------------------
cpjnybv7021rv    1F7059E0        SELECT 1 FROM tt WHERE rownum<2
cpjnybv7021rv    1F7059E0        SELECT 1 FROM tt WHERE rownum<2
```

可以看出，这两个 SQL 被认为是同一个 SQL，但是 cpjnybv7021rv 包含两个子游标。为什么会产生两个子游标呢？通过 v$sql_shared_cursor，可以看到：

```
SQL> SELECT sql_id,address,child_address,child_number,translation_mismatch FROM
  2  v$sql_shared_cursor WHERE sql_id='cpjnybv7021rv';

SQL_ID        ADDRESS  CHILD_AD CHILD_NUMBER T
------------- -------- -------- ------------ -
cpjnybv7021rv 1F7059E0 231974B8            0 N
cpjnybv7021rv 1F7059E0 232BC270            1 Y
```

第一个子游标在这个视图中所有的 mismatch 都是 N，第二个 SQL 是由 translation_mismatch 而导致不能共享，其原因是在做事务时发现相关的对象不同。通过库 Cache DUMP 看：

```
BUCKET 67323:
  LIBRARY OBJECT HANDLE: handle=1f7059e0 mutex=1F705A94(2)
```

```
name=SELECT 1 FROM tt WHERE rownum<2
hash=1cd693ce0a964189cac69e5ece0106fb timestamp=12-11-2007 16:18:05
namespace=CRSR flags=RON/KGHP/TIM/KEP/PN0/SML/KST/DBN/MTX/[120100d4]
kkkk-dddd-llll=0001-0001-0001 lock=0 pin=0 latch#=3 hpc=0000 hlc=0000
lwt=1F705A3C[1F705A3C,1F705A3C] ltm=1F705A44[1F705A44,1F705A44]
pwt=1F705A20[1F705A20,1F705A20] ptm=1F705A28[1F705A28,1F705A28]
ref=1F705A5C[1F705A5C,1F705A5C] lnd=1F705A68[232C803C,1F705164]
  DEPENDENCY REFERENCES:
  reference latch flags
  --------- ----- -------------------
   206b0c68     0 [60]
   20642c5c     0 [60]
  LIBRARY OBJECT: object=206b138c
  type=CRSR flags=EXS[0001] pflags=[0000] status=VALD load=0
  CHILDREN: size=16
  child#    table reference   handle
  ------ -------- --------- --------
       0 206b1318  206b0fcc 231974b8
       1 206b1318  206b112c 232bc270
  DATA BLOCKS:
  data#     heap  pointer    status pins change whr
  ----- -------- -------- --------- ---- ------ ---
      0 2316d4bc 206b1424 I/P/A/-/-    0 NONE   00
BUCKET 67323 total object count=1
```

这个父游标包含两个子游标，其中一个是 231974b8。下面我们来看看这个游标：

```
LIBRARY OBJECT HANDLE: handle=231974b8 mutex=2319756C(0)
namespace=CRSR flags=RON/KGHP/PN0/[10010000]
kkkk-dddd-llll=0000-0041-0041 lock=0 pin=0 latch#=3 hpc=0000 hlc=0000
lwt=23197514[23197514,23197514] ltm=2319751C[2319751C,2319751C]
pwt=231974F8[231974F8,231974F8] ptm=23197500[23197500,23197500]
ref=23197534[206B0FCC,206B0FCC] lnd=23197540[23197540,23197540]
  CHILD REFERENCES:
  reference latch flags
  --------- ----- -------------------
   206b0fcc     0 CHL[02]
  LIBRARY OBJECT: object=206b0b2c
  type=CRSR flags=EXS[0001] pflags=[0000] status=VALD load=0
  DEPENDENCIES: count=1 size=16
  dependency#    table reference   handle position flags
  ----------- -------- --------- -------- -------- -------------------
            0 20759070  20758d70 230c63b8       14 DEP[01]
  READ ONLY DEPENDENCIES: count=1 size=16
  dependency#    table reference   handle flags
  ----------- -------- --------- -------- -------------------
            0 206b0ee8  206b0c68 1f7059e0 /ROD/KPP[60]
  ACCESSES: count=1 size=16
  dependency# types
  ----------- -----
            0 0009
  TRANSLATIONS: count=1 size=16
  original    final
  -------- --------
```

```
    230c63b8 230c63b8
    DATA BLOCKS:
    data#     heap  pointer    status pins change whr
    ----- -------- -------- --------- ---- ------ ---
        0 23202038 206b0c7c I/-/A/-/-    0 NONE   00
        6 20758ca4 201a185c I/-/A/-/-    0 NONE   00
```

下面来看看 230c63b8 对应的对象：

```
BUCKET 99929:
  LIBRARY OBJECT HANDLE: handle=230c63b8 mutex=230C646C(0)
  name=SCOTT.TT
  hash=dcfddf221c9799c3b07c3d16af4b8659 timestamp=12-11-2007 16:16:45
  namespace=TABL flags=KGHP/TIM/SML/[02000000]
  kkkk-dddd-llll=0000-0701-0701 lock=N pin=0 latch#=1 hpc=0002 hlc=0002
  lwt=230C6414[230C6414,230C6414] ltm=230C641C[230C641C,230C641C]
  pwt=230C63F8[230C63F8,230C63F8] ptm=230C6400[230C6400,230C6400]
  ref=230C6434[230C6434,230C6434] lnd=230C6440[232B9660,1F6575D8]
    DEPENDENCY REFERENCES:
    reference latch flags
    --------- ----- -------------------
     20478ce0     2 DEP[01]
     20758d70     0 DEP[01]
    LOCK OWNERS:
        lock     user  session count mode flags
    -------- -------- -------- ----- ---- ------------------------
    20f9e5e4 2373238c 23733654     0 N    [4000]
    20fc86cc 2373238c 2373238c     0 N    [4000]
    LIBRARY OBJECT: object=20758884
    type=TABL flags=EXS/LOC[0005] pflags=[0000] status=VALD load=0
    DATA BLOCKS:
    data#     heap  pointer    status pins change whr
    ----- -------- -------- --------- ---- ------ ---
        0 2324da64 2075891c I/-/A/-/-    0 NONE   00
        8 20758aac 206b7298 I/-/A/-/-    0 NONE   00
        9 20758b44 204cf62c I/-/A/-/-    0 NONE   00
       10 20758b94 2073e4e0 I/-/A/-/-    0 NONE   00
  BUCKET 99929 total object count=1
```

再来看看第二个子游标，即 232bc270：

```
LIBRARY OBJECT HANDLE: handle=232bc270 mutex=232BC324(0)
  namespace=CRSR flags=RON/KGHP/PN0/[10010000]
  kkkk-dddd-llll=0000-0041-0041 lock=0 pin=0 latch#=3 hpc=0000 hlc=0000
  lwt=232BC2CC[232BC2CC,232BC2CC] ltm=232BC2D4[232BC2D4,232BC2D4]
  pwt=232BC2B0[232BC2B0,232BC2B0] ptm=232BC2B8[232BC2B8,232BC2B8]
  ref=232BC2EC[206B112C,206B112C] lnd=232BC2F8[232BC2F8,232BC2F8]
    CHILD REFERENCES:
    reference latch flags
    --------- ----- -------------------
     206b112c     0 CHL[02]
    LIBRARY OBJECT: object=20642b20
    type=CRSR flags=EXS[0001] pflags=[0000] status=VALD load=0
    DEPENDENCIES: count=1 size=16
    dependency#    table reference   handle position flags
```

```
----------- -------- --------- -------- -------- -------------------
          0 2059d22c  2059cf2c 1f708c18       14 DEP[01]
READ ONLY DEPENDENCIES: count=1 size=16
dependency#    table reference   handle flags
----------- -------- --------- -------- -------------------
```

看看 1f708c18 对应的对象：

```
BUCKET 68671:
  LIBRARY OBJECT HANDLE: handle=1f708c18 mutex=1F708CCC(0)
  name=T2.TT
  hash=b6efdd23a914ea2ea1b8ffeb7d170c3f timestamp=12-11-2007 16:16:15
  namespace=TABL flags=KGHP/TIM/SML/[02000000]
  kkkk-dddd-llll=0000-0701-0701 lock=N pin=0 latch#=2 hpc=0002 hlc=0002
  lwt=1F708C74[1F708C74,1F708C74] ltm=1F708C7C[1F708C7C,1F708C7C]
  pwt=1F708C58[1F708C58,1F708C58] ptm=1F708C60[1F708C60,1F708C60]
  ref=1F708C94[1F708C94,1F708C94] lnd=1F708CA0[1F705910,1F708B48]
    DEPENDENCY REFERENCES:
    reference latch flags
    --------- ----- -------------------
     207a6154     2 DEP[01]
     2059cf2c     0 DEP[01]
    LOCK OWNERS:
        lock     user  session count mode flags
    -------- -------- -------- ----- ---- ------------------------
    2125a5dc 2373491c 23733654     0 N    [4000]
    212345d0 2373491c 2373491c     0 N    [4000]
    LIBRARY OBJECT: object=2059ca40
    type=TABL flags=EXS/LOC[0005] pflags=[0000] status=VALD load=0
    DATA BLOCKS:
    data#     heap  pointer    status pins change whr
    ----- -------- -------- --------- ---- ------ ---
        0 2324abc0 2059cad8 I/-/A/-/-    0 NONE   00
        8 2059cc68 2059c8fc I/-/A/-/-    0 NONE   00
        9 2059cd00 2059c10c I/-/A/-/-    0 NONE   00
       10 2059cd50 2059bd14 I/-/A/-/-    0 NONE   00
  BUCKET 68671 total object count=1
```

除了这种情况，还有哪些不可共享的原因呢？从 v$sql_shared_cursor 中就可以看出mismatch 的各种原因。

另外一种典型的 SQL 不能共享的条件是两个 SQL 的执行计划不同。如果两个 SQL 相同，但是由于参数不同，必须使用不同的执行计划，那么这种 SQL 最好还是不要共享。因为 SQL 共享带来的好处可能只是执行了错误的执行计划的几百分之一。从这方面来看，我们也不能片面地强调 SQL 共享，而忽略了由于 SQL 共享带来的问题。

举个简单的例子，比如 ta 表有个字段是 STATUS，其中 99%的值都是 END，只有 1%的值是 BEGIN。我们的大多数程序都是每次读取值为 STATUS='BEGIN'的行，然后处理，处理结束后设置 STATUS='END'，只有少量的统计操作需要查询 STATUS='END'的行的数量，这种情况下如果使用柱状图，优化器就能使用合理的执行计划，因此这种情况下不能使用绑定变量。如果使用了绑定变量，那么如果是早期版本，优化器不会使用柱状图，而使用默认的选择性值来判断。如果

是 Oracle 9i 或者以后版本的数据库，绑定变量窥视（bind value peeking）技术可以通过使用柱状图来选择较好的执行计划。但是绑定变量窥视只是在 SQL 第一次被执行、执行硬分析时进行，后面的所有 SQL 由于使用了绑定变量，会被认为是安全的，不会进行窥视。这个 SQL 可能有两种执行计划，走索引或者全表扫描。而选择哪种执行计划，完全取决于做硬分析的那个 SQL 的绑定变量的值，而无法使优化器做出最佳选择。在这种情况下，共享 SQL 的代价就太大了。因此我们在学习 Oracle 时一定要活学活用，而不能教条主义。使用绑定变量在绝大多数情况下是对的，而在这种情况下，绝对不能使用绑定变量。

6月3日 拆分大型 SQL

这几天在做账务处理，需要将一批数据生成到临时表里，然后再进行处理。这个操作以往就十分慢。这个月的数据量比平时大 20%左右，昨天晚上跑了 6 个小时还没有结果，后来没办法就中断了。负责这个业务的小叶问我有没有什么办法进行优化。我看了一下，是一个 CTAS 的语句，将数据生成到一张临时表里：

```
   CREATE TABLE JFQR_TEMP
     TABLESPACE TS_SRHZ
     AS
     SELECT /*+ full(A) PARALLEL(A,4) */ A.YWID ZWYWID,C.YWID,A.SRXM,A.ZMID,
     DECODE(A.WJBZ,'APS',H.ZMLX,D.ZMLX) ZMLX,DECODE(A.WJBZ,'APS','1',D.FYLB) FYLB,
     DECODE(D.PARTNER_ID,1001,'1',1002,'2',1003,'3',2001,'4') ZHLX,
     DECODE(SUBSTR(A.ZMID_JC,1,1),'R',to_char(A.ZMID),A.ZMID_JC) ZMID_JC,
     DECODE(A.WJBZ,'APS',H.SUM_TYPE,DECODE(NVL(E.SUM_TYPE,D.SUM_TYPE),'10','长话',
'20','市话','40','数据','其他')) SUM_TYPE,
     NVL(SUM(A.JFJE)/100,0) YSJE,
     NVL(SUM(A.JFJE)/100,0) ZSJE,C.YWHM,A.SBLX
YWLX,DECODE(A.GHLX,'1','8','2','7','3','5',A.GHLX)
GHLX,DECODE(A.KHLX,'1','1','2','3','3','7','4','Z',A.KHLX)
FLSX,A.YHLX,A.XTBH,A.HMZT,A.YYSLX,A.WJBZ,DECODE(ZQ,0,'${LoadTimeThis}',G.BILLING_M
ONTH) ZQ,qrxm,jflx
     FROM JFHZGW_YSSJ_ A,
     JFYWQYW_ C,
     JFZMQYW D,
     JFZMQYW E,
     JCZC.JFZQ G,
     JFFV_APS_ZW H
     WHERE A.YWID=C.ZWYWID(+) AND
           A.ZMID=D.ACCT_ITEM_TYPE_ID(+) AND
DECODE(SUBSTR(A.ZMID_JC,1,1),'R',to_char(A.ZMID),A.ZMID_JC)=to_char(E.ACCT_ITEM_
TYPE_ID(+)) AND
           A.ZQ=G.BILLING_CYCLE_ID(+) AND
           A.ZMID=H.JFLB(+)
     GROUP BY
A.YWID,C.YWID,A.SRXM,A.ZMID,DECODE(A.WJBZ,'APS',H.ZMLX,D.ZMLX),D.FYLB,D.PARTNER_ID,
DECODE(SUBSTR(A.ZMID_JC,1,1),'R',to_char(A.ZMID),A.ZMID_JC),DECODE(A.WJBZ,'APS',H.
SUM_TYPE,DECODE(NVL(E.SUM_TYPE,D.SUM_TYPE),'10','长话','20','市话','40','数据','其他')),
C.YWHM,A.SBLX,A.GHLX,A.KHLX,A.YHLX,A.XTBH,A.HMZT,A.YYSLX,A.WJBZ,DECODE(ZQ,0,
'${LoadTimeThis}',G.BILLING_MONTH),qrxm,jflx;
```

从这个 SQL 看相当复杂，而且涉及多张很大的表。这个 SQL 如果不拆分，纯粹想通过调整

执行计划来进行优化将十分困难。我建议她将 SQL 拆分了，用一个存储过程来实现。先扫描那张大表，然后通过访问几张维表进行数据过滤和表连接，最终将结果插入到临时表中。为了提高插入速度，可以在其中使用 bulk 操作。小叶没有处理过类似的案例，因此希望我给她一个更明确的指导。我考虑一两句话也说不明白，所以就写了一个样板程序给她：

```
create or replace procedure testall
is
  TYPE TOWNER    IS TABLE OF             VARCHAR2(30)    INDEX BY BINARY_INTEGER;
  TYPE TSEGMENT_NAME    IS TABLE OF      VARCHAR2(81)    INDEX BY BINARY_INTEGER;
  TYPE TSEGMENT_TYPE    IS TABLE OF      VARCHAR2(18)    INDEX BY BINARY_INTEGER;
  TYPE TFILENAME    IS TABLE OF          VARCHAR2(100)   INDEX BY BINARY_INTEGER;
  TYPE TRFILENAME    IS TABLE OF         VARCHAR2(100)   INDEX BY BINARY_INTEGER;
  TYPE TFILE_ID      IS TABLE OF NUMBER  INDEX BY BINARY_INTEGER;
  TYPE TRFILE_ID     IS TABLE OF NUMBER  INDEX BY BINARY_INTEGER;
  VOWNER TOWNER;
  VSEGMENT_NAME TSEGMENT_NAME;
  VSEGMENT_TYPE TSEGMENT_TYPE;
  VFILE_ID TFILE_ID;
  VRFILE_ID TRFILE_ID;
  VFILENAME TFILENAME;
  VRFILENAME TRFILENAME;
  TYPE TFILELIST1 IS TABLE OF varchar2(100) INDEX BY BINARY_INTEGER;
  TYPE TFILELIST2 IS TABLE OF varchar2(100) INDEX BY BINARY_INTEGER;
  VFILELIST1 TFILELIST1;
  VFILELIST2 TFILELIST2;
  CURSOR CFILELIST1 IS SELECT FILE_ID,FILENAME FROM TB1;
  CURSOR CFILELIST2 IS SELECT relative_fno,FILENAME FROM TC1;
  str varchar2(100):='SELECT OWNER,SEGMENT_NAME,SEGMENT_TYPE,FILE_ID,RELATIVE_FNO
FROM TA1';
  rows1 integer;
  rows_total integer;
  rows2 integer;
  i integer;
  j integer;
  k integer;
  type my_cur_type is ref cursor;
  C3 my_cur_type;
BEGIN
     DBMS_OUTPUT.ENABLE(1000000);
----装载维表数据
    FOR REC IN CFILELIST1 LOOP
      VFILELIST1(REC.FILE_ID):=REC.FILENAME;
    END LOOP;
    FOR REC IN CFILELIST2 LOOP
      VFILELIST2(REC.relative_fno):=REC.FILENAME;
    END LOOP;
    OPEN  C3 FOR STR;
    rows_total:=0;
    rows1:=rows_total;
    loop
----批量提取数据到数组，每次最多提取 1000 条
      FETCH c3 BULK COLLECT INTO VOWNER,VSEGMENT_NAME,VSEGMENT_TYPE,VFILE_ID,VRFILE_ID
```

```
limit 1000;
----提取出的总的数据量（注意是总数，不是每次独立提取出的数据量）
      rows_total:=c3%bulk_rowcount(1);
----rows2 里是本次提取的总数，然后将 rows1 设置为当前总数
      rows2:=rows_total-rows1;
      rows1:=rows_total;
----如果本次提取出的数据量为 0，说明记录已经全部提取完，退出
      exit when rows2=0;
----通过维表的数据补充相关列的值
       for k in 1..rows2 loop
         vfilename(k):=vfilelist1(vfile_id(k));
         vRfilename(k):=vfilelist2(vrfile_id(k));
       end loop;
      dbms_output.put_Line('begin to insert:'||to_char(rows2)||' rows');
      i:=1;
      j:=rows2;
      dbms_output.put_line('ij=('||to_char(i)||','||to_char(j)||')');
----使用 FORALL INSERT，批量插入数据
      FORALL k in i..j
        INSERT into ta2
values(VOWNER(k),VSEGMENT_NAME(k),VSEGMENT_TYPE(k),VFILENAME(K),VRFILENAME(K));
      commit;
----如果 rows2 小于 1000，说明记录已经全部提取完，退出
      exit when rows2<1000;
  end loop;
    close C3;
    commit;
END;
/
```

根据我提供的脚本，小叶很快改好了这个程序。测试了一下，居然不到 1 个小时就跑出来了。找出几个数据核对了一下，也完全正确。这个往常要跑两三个小时，昨天晚上跑了 6 个小时都没跑出来的程序，居然有这么好的优化效果，大家都很高兴。

优化小技巧　拆分复杂 SQL

拆分复杂 SQL 是 SQL 优化中经常会碰到的一个工作。开发人员由于在开发过程中往往只关注于 SQL 功能的实现，而不关心其性能，所以在编写 SQL 时经常会写出十分复杂的查询。一旦某个 SQL 写得十分复杂，就有可能消耗过多的系统资源。实际上，开发人员是对业务逻辑最为熟悉的人员，他们知道一个业务应该先访问哪张表，取出什么数据，然后再进行什么操作，如果他们能够把这个流程用一个存储过程写出来，那么程序将会十分清晰，而且不会出现性能问题。而事实上，一旦碰到类似的问题，程序员往往会绞尽脑汁，写出一个十分复杂的 SQL 出来，就像今天我们碰到的那个 SQL 一样。这个 SQL 往往会消耗大量的系统资源，而且一旦系统上线后，就没有人敢去修改它。随着时间的推移，甚至开发这个 SQL 的人都很难看懂其中的业务逻辑关系，今后再去优化这个 SQL 的难度就越来越大了。

因此作为 DBA，总是希望开发人员不要写过于复杂的 SQL，最好能使用存储过程来处理业务逻辑。但是我们不得不面对越来越多的复杂 SQL。在这种情况下，我们需要了解拆分大型 SQL

的技术。

拆分大型 SQL 首先要理解 SQL 的含义，把 SQL 的功能分解为多个单元，进而划分出其中的中间数据集。如果通过对这些中间数据集的分析，发现这个 SQL 从业务逻辑上来看是可以分步执行的，那么这个 SQL 就是可以拆分的。

拆分 SQL 最好能从业务的角度来做，因此最好能够得到开发人员的支持。如果得不到开发人员的支持，必须首先确保对 SQL 功能的分析是完全正确的，否则可能会吃力不讨好。

在拆分 SQL 时，可能需要使用一些复杂的技术，比如 Hash 连接技术。有时候，需要我们通过类似 Oracle 的散列连接的算法对多个结果集进行合并。如果拥有现成的类库，可以直接调用，否则对于一般程序员来说是很难处理的。如果碰到这种情况，可以首先将这两个结果集写入临时表，然后通过 SQL 对这两张临时表使用 Hash 连接进行连接。

6月4日 TNS-12535

今天是周日，我让老于抓紧时间回趟家，我来值一下班。由于这几天一直在做账务处理，所以周末办公室里也还是有几个甲方的人在做处理。由于不是工作日，所以也没有去得太早，早上吃完早饭后上了会网，到办公室时已经是 10 点多了。刚到办公室就接到一个电话，他们在把一批数据用 SQL * Loader 导入到数据库时出现了一个问题。SQL*Loader 报错 LDR-704，ORA-12170 连接超时出现，导入失败。导入的数据量并不是很大，也就是 100 多万条记录。每次导入进行了一段时间后就会报错，每次报错的时间点都不同，并没有一定的规律性。

这个数据装载功能是新业务系统引入的，今天是第一次运行。我让他们马上打开客户端的跟踪。通过 Trace 文件来分析一下。将 trace_level_client 设置为 16 后，又进行了一次数据载入操作。打开 Trace 文件后，看到如下信息：

```
(5580) [7-11 月-2008 14：07:19:324] nttcnp: creating a socket.
(5580) [7-11 月-2008 14：07:19:324] nttcnp: exit
(5580) [7-11 月-2008 14：07:19:324] nttcni: entry
(5580) [7-11 月-2008 14：07:19:324] nttcni: trying to connect to socket 3804.
(5580) [7-11 月-2008 14：07:40:351] ntt2err: entry
(5580) [7-11 月-2008 14：07:40:351] ntt2err: soc 3804 error - operation=1, ntresnt[0]=505,
ntresnt[1]=60, ntresnt[2]=0
(5580) [7-11 月-2008 14：07:40:351] ntt2err: exit
(5580) [7-11 月-2008 14：07:40:445] nttcni: exit
(5580) [7-11 月-2008 14：07:40:445] nttcon: exit
(5580) [7-11 月-2008 14：07:40:445] nserror: entry
(5580) [7-11 月-2008 14：07:40:445] nserror: nsres: id=0, op=65, ns=12535, ns2=12560;
nt[0]=505, nt[1]=60, nt[2]=0; ora[0]=0, ora[1]=0, ora[2]=0
(5580) [7-11 月-2008 14：07:40:445] nsopen: unable to open transport
```

在 Trace 文件里可以明确地看到：trying to connect to socket 3804,soc 3804 error - operation =1,ntresnt[0]=505,ntresnt[1]=60,ntresnt[2]=0。从信息上看，是打开套接字 3804 时出的问题。主错误信息是 TNS-505，也就是操作超时，说明是连接 3804 端口时出现了超时。ntresnt[1]=60，这是操作系统上的错误信息。由于客户端是 Windows Server 2003，所以通过下面的命令可以查看操作系统的错误信息：

```
C:\>net helpmsg 60
远程适配器不兼容。
```

从上面的情况看，可能是网络出现了问题。从后面的信息 nserror: nsres: id=0, op=65, ns=12535, ns2=12560; nt[0]=505, nt[1]=60, nt[2]=0; ora[0]=0, ora[1]=0, ora[2]=0

可以看出，实际上内部报的错误号是TNS-12535。这个错误是在复杂网络环境下，特别是穿透防火墙时经常会出现的故障。因此，我猜测是由于通过了防火墙，导致了套接字重定向时被防火墙挡住了，导致连接3804端口失败。

由于今天是周末，网络管理员没有上班，所以找了一大圈才找到一个对网络比较了解的人。通过那个人的描述，我们发现在那台Windows 2003服务器到计费服务器需要经过2道防火墙。网络管理人员也说不清楚到底防火墙的策略是什么，不过可以确认的是，其中一道防火墙是计费网络和业务网络之间的隔离设备，使用了NAT。

TNS-12535在NAT环境下出现是很常见的，因此基本可以定位由于防火墙导致了这个问题。由于目前没办法在短时间内解决网络问题，所以我建议他们把数据复制到服务器上，然后再导入。将数据FTP过来以后再执行导入，不到10分钟就全部导入完毕了。

今日点评

通过防火墙访问数据库是很常见的问题。不过通过防火墙访问数据库经常会碰到不稳定的现象，就是经常出现一些莫名其妙的ORA/TNS错误，连接会突然中断。这主要是由于防火墙的包过滤和NAT功能引起的。处理这样的情况时，一般可以采取以下措施：

- 通过VPN穿透防火墙，如果网络稳定，这是最推荐的方式；
- 使用CMAN穿透防火墙，避免端口重定向；
- 在防火墙中加载Oracle SQL*Net模块（部分防火墙支持）。

关于这方面的详细信息，可以参考以下文档。

- Note: 152988.1 How to Configure Oracle Net to Access a Server Behind a Firewall。
- Note: 152133.1 How To Do Oracle Net Connections Over a Firewall?
- Note: 45226.1 SQL*Net and Firewalls。
- Note: 2084440.6 Oracle and Firewalls: Answers to Frequently Asked Questions。
- Note: 125021.1 Oracle Connections and Firewalls。
- Note: 397393.1 External Clients behind NAT Translation / Port Forwarding / Tunneling Fail to Connect to Database (ORA-12535 or ORA-12541)。
- Note: 558959.1 Oracle Connection Manager Configuration。

优化小技巧　防火墙和SQL*Net

很多防火墙都有包过滤和NAT功能。对于使用端口重定向的SQL*Net协议来说，可能会碰到很多问题，导致SQL*Net通过防火墙访问出现故障。为了解决这个问题，一般来说有几种常用方案。

- 使用VPN：如果客户端能够通过VPN拨入内网，那么就不需要使用NAT了，而且也不会受到包过滤的限制。

- 使用 CMAN（Oracle Connect Manager）。
- 在防火墙中加载 Oracle 专用模块（部分防火墙支持）。

通过 VPN 是最为简单的解决方案，完全通过网络技术实现，不涉及 Oracle 的配置。还有一种方法是使用 CMAN，配置 SQL*Net 代理。CMAN 代理和客户端之间是固定连接，连接后不会再做端口重定向，这样就不会出现类似于今天出现的这种问题。

注意，关于 CMAN 安装与配置，请参考 Metalink 文档 2077721.6 Installation and Configuration of Net8 Connection Manager (CMAN)（针对 Oracle 10g，请参考 298916.1 A Guide to 10g CMAN Configuration）。

优化小技巧 跟踪 SQL*Net

碰到 SQL*Net 问题，怎么办呢？大多数情况下，SQL*Net 问题无法通过 Oracle 数据库的日志进行分析。我们首先根据服务器端的监听状态、监听日志和服务器端的 sqlnet.log 来进行初步的分析。这些日志都存放在$ORACLE_HOME/network/log 目录下。sqlnet.ora 文件的格式如下：

```
Fatal NI connect error 12638, connecting to:
 (DESCRIPTION=(LOCAL=YES)(ADDRESS=(PROTOCOL=beq)))

  VERSION INFORMATION:
   TNS for 32-bit Windows: Version 9.2.0.8.0 - Production
   Oracle Bequeath NT Protocol Adapter for 32-bit Windows: Version 9.2.0.8.0 -
Production
  Time: 15-12 月-2008 08:05:23
  Tracing not turned on.
  Tns error struct:
    ns main err code: 12638
    TNS-12638: 身份证明检索失败
    ns secondary err code: 0
    nt main err code: 0
    nt secondary err code: 0
    nt OS err code: 0
```

在这个 Trace 信息里，找到 TNS 的错误信息，根据 TNS-xxxx 就可以进行分析了。如果在服务器端的 Trace 里找不到有价值的信息，或者无法明确定位故障点，那么就需要通过客户端的 Trace 来进行分析。设置客户端 Trace 的方法如下：

修改客户端的$ORACLE_HOME/network/admin/sqlnet.ora

```
trace_level_client = <级别>
trace_file_client = client
trace_directory_client = <路径>
```

在今天分析 SQL*Net 故障时，我设置了 `trace_level_client =16`。在使用 16 级 Trace 时，会生成十分详细的跟踪信息。比如，我们生成的 Trace 文件为：

```
(7288) [10-11 月-2008 20:54:56:552] nsopen: opening transport...
(7288) [10-11 月-2008 20:54:56:552] nttcon: entry
(7288) [10-11 月-2008 20:54:56:552] nttcon: toc = 1
```

```
(7288) [10-11 月-2008 20:54:56:552] nttcnp: entry
(7288) [10-11 月-2008 20:54:56:552] nttcnp: creating a socket.
(7288) [10-11 月-2008 20:54:56:552] nttcnp: exit
(7288) [10-11 月-2008 20:54:56:552] nttcni: entry
(7288) [10-11 月-2008 20:54:56:552] nttcni: trying to connect to socket 3804.
(7288) [10-11 月-2008 20:54:56:552] snlinGetNameInfo: entry
(7288) [10-11 月-2008 20:54:56:552] snlinGetNameInfo: Using numeric form of host's
address 10.2.0.39
(7288) [10-11 月-2008 20:54:56:552] snlinGetNameInfo: exit
(7288) [10-11 月-2008 20:54:56:552] nttcni: connected on ipaddr 10.2.0.39
(7288) [10-11 月-2008 20:54:56:552] nttcni: exit
(7288) [10-11 月-2008 20:54:56:552] nttcon: NT layer TCP/IP connection has been
established.
(7288) [10-11 月-2008 20:54:56:552] nttcon: set TCP_NODELAY on 3804
(7288) [10-11 月-2008 20:54:56:552] nttcon: exit
```

上面是一次成功的连接，最终成功地打开了套接字 3804，并且进行了正常的通信。与今天日记里那个失败的结果相比较，就很容易搞清楚我们该如何分析这个 Trace 文件了。

6月5日 休息时总是最忙的

由于明天要进行实施，所以今天安排轮流休息，我被安排在上午休息。上午吃过早饭后回到房间躺了一会儿，不过总是睡不着，我没有白天睡觉的习惯。躺了半天，精神头越来越好，实在无聊只好起来上网。刚刚在 QQ 上和群里的哥们聊了几句，一个客户通过 MSN 找到我了。他说有个很奇怪的操作，系统其他的都很正常，就是有一个存储过程，只要一编译就挂住。我问他是不是比较慢，他说不是慢就像挂死了一样，根本没有反应。

碰到这种情况，一般来说首先应该做一个 hanganalyze。于是我建议他马上用系统账号登录，然后做一个 hanganalyze：

```
SQL>oradebug setmypid;
SQL>oradebug hanganalyze 3;
```

很快他就把 Trace 文件发给我了。打开 Trace 文件，可以很明显看到阻塞的情况：

```
==============
HANG ANALYSIS:
==============
Open chains found:
Chain 1 : <sid/sess_srno/proc_ptr/ospid/wait_event> :
    <48/29750/0x3ed60060/7856/PX Deq: Join ACK>
 -- <141/31823/0x3ed5d680/2291/library cache pin>
Chain 2 : <sid/sess_srno/proc_ptr/ospid/wait_event> :
    <76/16377/0x3ed554b0/6507/No Wait>
Chain 3 : <sid/sess_srno/proc_ptr/ospid/wait_event> :
    <98/8617/0x3ed65c80/4550/No Wait>
Chain 4 : <sid/sess_srno/proc_ptr/ospid/wait_event> :
    <117/43613/0x3ed55080/6505/No Wait>
```

其中 sid 为 141 的会话就是那个被挂起的会话。从 hanganalyze 报告上看，这个会话被 sid 为 48 的会话阻塞了。48 号会话正在等待 PX:Deq ACK。从 v$session 中，我发现 48 号会话的状态是 killed，看样子这个会话已经被杀掉了。在 v$session 中继续检查，发现这个会话的程序是 Oracle@yzstat (P001)。从现象上看，这是由于并行查询被终止后，slaver 被杀掉，会话已经被标志为 killed，不过进程并没有退出，资源也就没有释放，而这个会话正好 PIN 住了 141 会话编译存储过程所需要的。确认了这个进程可以杀掉后，使用 kill-9 7856 杀死该进程，被阻塞的操作马上完成了。

处理完了以后，又和客户讲了半天 hanganalyze 的使用方法以及注意事项。处理完这个案例后，看看时间已经是 11 点多了，该去和老于约好的餐厅了。我到的时候，老于他们几个都到了。

下午我和老熊值班，老于和老肖回去休息。大家一起讨论了明天的主要事情，都对明天的调整表示出了很强的信心，最好是能够毕其功于一役，这样我们今后的工作也比较好开展。如果系统级的调整达到了目标，下一阶段重点和集成商一起进行 SQL 优化就可以了。

下午过得较为平淡，系统负载很高，特别是 I/O 的压力很大，不过总的来说还算平稳。这几天营业系统负载较轻，主要的系统压力来自于账务处理。我和老熊也在抓紧时间整理文档。主要的 SQL 都已经整理出来了，并且绝大部分 SQL 的优化方案也已经细化，就等开发商去修改程序了。今天我们两个人对每个 SQL 的状态按照最新的反馈情况进行了更新，更新后重新统计了一下，发现完成状态的 SQL 只有 20%多一点，而且除了一个 UNION 改为 UNION ALL 的语句和几个我们协助甲方修改的存储过程外，其他完成优化的语句基本上都是通过调整索引来优化的。也就是说，开发商调整程序的进度十分缓慢，我和老熊都感觉到，开发商修改程序的进度可能会跟不上整个优化的进度。

碰到这种情况，我们唯一能做的就是把优化方案搞得更为细致，使开发商找不出不修改程序的理由。因此我和老熊分了分工，每个人负责几个复杂 SQL 的分析，争取在这几天里把所有 SQL 的优化方案都写得比较明确，以便提交给开发商后，他们可以直接修改程序，而不需要再去做分析了。

不知不觉已经快 6：00 了。一般来说，5：30 就下班了，不过 5：30 之后的 15 分钟里可能会出现一个小的业务高峰，因为所有的营业员下班前都要做日常扎账的功能。我临走前看了看 OEM 的监控屏幕，除了 I/O 高一点，CPU 的负载已经从刚才高峰期间的 100%下降到 70%左右，看样子还算稳定。现在已经快 6：00 了，各个营业厅应该已经下班了，优化前的倒数第二个营业日应该是平稳过去了。收拾好电脑，我和老熊正准备离开。同样准备离开的小孙突然叫了起来："白哥，你快看看，数据库是不是宕了？"

我马上登上系统，发现数据库确实是宕了。从日志上看到下面的信息：

```
Sun May  5 18:02:34 2006
Errors in file /data1/Oracle9/app/admin/sdh/bdump/sdh_dbw3_462948.trc:
ORA-27061: skgfospo: waiting for async I/Os failed
IBM AIX RISC System/6000 Error: 22: Invalid argument
Sun May  5 18:02:34 2006
DBW3: terminating instance due to error 27061
Instance terminated by DBW3, pid = 462948
```

第一眼看到这个信息，就知道这是异步 I/O 的问题。在 AIX 上，由于异步 I/O 引起的宕机也不是什么新鲜事了。由于操作系统和数据库的 bug 问题，如果数据库使用了文件系统，在 AIX 上很容易出现由于异步 I/O 参数设置不足而导致的系统宕机。因此，我第一时间检查了 aio 的设置：

```
# lsattr -El aio0
autoconfig available STATE to be configured at system restart True
fastpath   enable    State of fast path                       True
kprocprio  39        Server PRIORITY                          True
maxreqs    4096      Maximum number of REQUESTS               True
```

```
maxservers 40        MAXIMUM number of servers per cpu        True
minservers 10        MINIMUM number of servers                True
```

果然所有的异步 I/O 参数都使用的是默认值。临时的解决方案是加大异步 I/O 相关参数，不过这也只能作为权宜之计。加大异步 I/O 参数只能减少宕机的机会，并不能彻底解决这个问题，必须做操作系统升级并且打数据库的补丁才能解决这个问题。不过如果明天晚上优化时加上这两个操作，整个优化计划都要推倒重做。从目前的情况来看，条件是不允许的。我和老于通了电话沟通了这件事情，大家的一致看法是，先通过调整参数顶一下，等系统稳定后再找时间升级。因为明天的优化会较大幅度地减少物理 I/O，另外通过加大异步 I/O 相关的参数也可以加大异步 I/O 的处理能力，经过调整后，由异步 I/O 产生的宕机现象将会得到控制。等优化完成后，下一个账期前再进行操作系统的升级和数据库补丁的安装。

和老于讨论完后，我给张工打了个电话，把今天宕机的原因告诉了他，并且把我们的想法也和他做了沟通，张工比较赞同我们的想法。

回到酒店，我和老于一起调整了优化方案，增加了对异步 I/O 参数调整的内容：

- 将 `Maxreqs` 修改为 8192；
- 将 `Maxservers` 修改为 256；
- 将 `Minservers` 修改为 50。

今日点评

今天老白碰到了系统挂住的情况，然后又碰到了一次由于异步 I/O 而引起的宕机。由于异步 I/O 引起宕机是 DBA 经常碰到的，一般的错误号都是 ORA-27XXX。在 AIX 下，导致异步 I/O 的问题在 Oracle 8i、Oracle 9i 和 Oracle 10g 中都有发生。一般在使用文件系统的环境下，使用裸设备就很少出现。因为在 AIX 下裸设备的异步 I/O 是核心支持的，而文件系统必须使用 AIO Server。碰到老白今天下午遇到的问题，一般来说需要将 AIX 升级到 5.3 TL6 或者更高的版本，并且打 Oracle 9i 的补丁。详情可以参考以下文档。

- Note:390656.1 I/O Interoperability Issue between IBM ML05 and Oracle Databases。
- Note:452866.1 Database crashes producing ORA-27061。
- Note:467282.1 Instance down due to ORA-27061: skgfospo: waiting for async I/Os failed。
- 4401119.8 Bug 4401119 - AIX: Crash possible on AIX (ORA-206)。

其实异步 I/O 的问题在其他操作系统下也常有发生。由于今天没有碰到，老白也就偷点懒了。大家可以去 Metalink 上查找相关的文档。

优化小技巧　使用 hanganalyze

数据库挂住是件比较头痛的事情。如何找到挂住的原因是 DBA 必须面临的课题。当数据库挂住时，大多数 DBA 往往是通过 `v$session_wait` 视图来进行分析。实际上，Oracle 有一个十

分有效的工具——hanganalyze。hanganalyze 可以十分清晰地将挂住的信息告诉 DBA，便于 DBA 进行进一步分析。hanganalyze 是从 Oracle 8i r2(8.1.6)开始提供的，其用法十分简单：

```
ALTER SESSION SET EVENTS 'immediate trace name HANGANALYZE level <level>';
```

或者

```
ORADEBUG hanganalyze <level>
```

比如：

```
sql>oradebug setmypid;
sql>oradebug hanganalyze 3;
```

对于<level>的定义，取值以及对应的说明如下。

- 10。所有进程（IGN state）。
- 5。Level 4 + 等待链中的所有进程（NLEAF state）。
- 4。Level 3 + 等待链中阻塞进程的状态（LEAF、LEAF_NW、IGN_DMP state）。
- 3。Level 2 + 被挂住的进程的状态（IN_HANG state）。
- 1 和 2。只输出 hanganalyze，不输出进程信息。

下面是一个 3 级的 hanganalyze（一般情况下，使用 3 级就可以了）：

```
/oracle/app/oracle/admin/bill/udump/ora_4550_bill.trcOracle8i
Enterprise Edition Release 8.1.7.4.0 - 64bit Production
With the Partitioning option
JServer Release 8.1.7.4.0 - 64bit Production
ORACLE_HOME = /oracle/app/oracle/product/8.1.7
System name: HP-UX
Node name: yzstat
Release: B.11.11
Version: U
Machine: 9000/800
Instance name: bill
Redo thread mounted by this instance: 2
Oracle process number: 75
Unix process pid: 4550, image: oracle@yzstat (TNS V1-V3)
*** SESSION ID:(98.8617) 2008-02-23 17:45:07.011***
2008-02-23 17:45:07.010
==============HANG ANALYSIS:==============
Open chains found:
Chain 1 :
<sid/sess_srno/proc_ptr/ospid/wait_event> :
<48/29750/0x3ed60060/7856/PX Deq: Join ACK>
-- <141/31823/0x3ed5d680/2291/library cache pin>
Chain 2 :
<sid/sess_srno/proc_ptr/ospid/wait_event> :
<76/16377/0x3ed554b0/6507/No Wait>
Chain 3 :
<sid/sess_srno/proc_ptr/ospid/wait_event> :
<98/8617/0x3ed65c80/4550/No Wait>
Chain 4 :
<sid/sess_srno/proc_ptr/ospid/wait_event> :
```

```
<117/43613/0x3ed55080/6505/No Wait>
Other chains found:
Extra information that will be dumped at higher levels:
[level  4] :   4 node dumps -- [LEAF] [LEAF_NW] [IGN_DMP]
[level  5] :   1 node dumps -- [NLEAF]
[level 10] :  78 node dumps -- [IGN]
State of nodes([nodenum]/sid/sess_srno/session/state/start/finish/[adjlist]/predecessor):
[0]/1/1/0x3ee65290/IGN/1/2//none
[1]/2/1/0x3ee65c10/IGN/3/4//none
[2]/3/1/0x3ee66590/IGN/5/6//none
[3]/4/1/0x3ee66f10/IGN/7/8//none
[4]/5/1/0x3ee67890/IGN/9/10//none
[5]/6/1/0x3ee68210/IGN/11/12//none
[6]/7/1/0x3ee68b90/IGN/13/14//none
[8]/9/28264/0x3ee69e90/IGN/15/16//none
[9]/10/45631/0x3ee6a810/IGN/17/18//none
[10]/11/53468/0x3ee6b190/IGN/19/20//none
[11]/12/1/0x3ee6bb10/IGN/21/22//none
......
*** WARNING: no system state dumped ***
====================END OF HANG ANALYSIS====================
```

我们可以看到，在 Open chains 里加粗部分就是可能挂住的环。对于确定的内部死锁，Oracle 会在 Cycles 里列出，比如：

```
Cycle 1 : <sid/sess_srno/proc_ptr/ospid/wait_event> :
   <980/3887/0xe4214964/24065/latch free>
 -- <2518/352/0xe4216560/24574/latch free>
 -- <55/10/0xe41236a8/13751/latch free>
```

我们可以看出，980 号会话在等待 latch free，它阻止了 2518 号会话和 55 号会话，这两个会话也在等待 latch free。这个时候杀掉 980 号会话就可以解开这个环。直接在 hanganalye 里就可以看到 980 号会话的 ospid 是 24065，直接在操作系统下使用 kill-9 24065 就可以了。

根据 hanganalyze 的结果，我们再去分析原因就容易多了。很多 DBA 喜欢通过脚本去分析 v$lock 和 v$session_wait。实际上，通过 hanganalyze 可以更为快捷地分析出系统挂住的原因。在做 hanganalyze 时，对于 Cycle，直接杀掉某个会话就可以解决问题。如果是 Open chains，那么就不一定是挂住了。系统如果比较慢，也会出现 Open chains。为了明确是由于系统比较慢导致的还是由于某个会话阻止了其他会话，可以做一个 hanganalyze，然后再等待 3～5 分钟做第二个 hanganalyze，两个比较一下就可以看得很清楚了。如果在两个 hanganalyze 中，chain 的状态没有任何改变，那么说明是挂住了，否则可能是由于慢导致的问题。

在最后，老白再给大家介绍一个小技巧，如果数据库挂住了，连登录都无法做了，我们该怎么处理呢？如果你比较幸运，使用的是 Oracle 10g，就可以通过下面的办法进行处理：

```
sqlplus -prelim /nolog
SQL>connect / as sysdba
SQL>oradebug setmypid;
SQL>oradebug hanganalyze 3
```

使用-prelim参数强行登录数据库，不过这个登录的会话只能做系统状态转储和hanganalyze

之类的诊断工作，不能做其他事情。不过这也为我们分析挂住的原因提供了很好的支持。记住这个小技巧是十分有用的。

优化小技巧　重做日志文件优化的要点

重做日志对 I/O 的影响十分大，特别是繁忙的系统。一般来说，我们都会建议数据库安装时，重做日志使用独立的盘组存放，而不要和其他文件混合存放。而在实际情况下，我们很少能够看到这样的用户。在早期的系统中，存储的容量有限，硬盘价格也比较贵，因此这些系统都没有采用重做日志文件独立存放。这样做最初的目的是为了节约宝贵的存储空间，而在磁盘价格十分低的今天，这种坏习惯被延续了下来。实际上，在硬盘价格相对便宜的今天，在系统规划之初，将重做日志文件存放在独立的系统上不是一件难事。另外，重做日志文件上的操作一般都是频繁的小 I/O，因此重做日志文件最好存放在 RAID 1 的存储设备上，而尽量不要存放在 RAID 5 上。由于重做日志文件是顺序写的，所以多个重做日志文件放在同一个磁盘上不会引起性能问题。不过同一个日志组的多个成员最好不要存放在相同的磁盘上，以避免 I/O 冲突。老白做过一个项目，把重做日志文件和数据文件分离后，数据文件所在盘组的 I/O 负载下降了 20%多。

不过如果你使用了重做日志镜像，也就是说一个重做日志组有多个成员，那么相同的成员最好不要存放在相同的磁盘上。在 10 年前，我经常使用一种存储方式，就是在系统中找两块裸盘，这两块盘在盘阵中做成 JBOD，把重做日志的两个镜像分别存放在这两块裸盘上。也就是说，重做日志文件存放在无 RAID 保护的独立盘上，依靠镜像来实现高可靠性。在 I/O 性能较差的年代，这种做法对于提高整体 I/O 性能帮助很大。目前 I/O 系统的能力越来越强，这种极端需求也很少出现了。并且现在绝大多数存储都不支持裸盘，所以这种配置现在基本上看不见了。不过对于 I/O 特别敏感的系统，为了提高安全性，必须设置重做日志镜像，那么也可以拿出两组独立的盘组，专门用于存放重做日志，这对于提升总体 I/O 性能的帮助很大，虽然这么做很浪费。

最后一点要说明的是，独立重做日志的存储一定要在数据库创建时就做，不要等到系统出现 I/O 问题时再去做。因为这时，DBA 可能会发现，盘阵中已经没有空闲的盘组给重做日志使用了。这也是最常见的现象。

重做日志的大小也会对系统性能有比较大的影响，包括对 I/O 系统。如果重做日志太小，会导致日志切换十分频繁，从而影响系统性能。一般来说，保证重做日志切换周期不低于 15 分钟是十分必要的。

6月6日 实施优化

今天晚上要做优化的实施，所以都睡了个懒觉。我没有睡懒觉的习惯，9：00不到就再也没法在床上趟着了，到楼下吃了早餐。吃完早餐后本来想出去走走，后来想了半天也没想好去哪，只好回到房间来上网。方案已经制订得十分完备了，实际操作主要由老肖来做，我只是负责协调，因此今天我的压力并不大。经过一个星期的突击，体力也确实有点透支了，今天晚上可能会遇到很多问题，也许并不会很顺利，因此我必须有足够的体力来保证晚上大脑不出现真空。10点多钟，大家纷纷起床了，然后一起在老于的房间里碰了一下，就去现场了。

客户对今天晚上的实施十分重视，张工他们三个都一起到现场来值守。预定的停机时间是晚上8：00。为了保证实施的顺利进行，提前一天申请了停止所有夜间Job和数据库备份作业。

从下午5：00开始，通过OEM的实时监控看到系统负载很小，因此我和老于决定对于主要业务表的分析操作提前开始。从5：00开始，我们首先挑选了部分比较小的表进行分析，这些小表很快都分析完了。于是老肖开始分析第二批略大的表，这批表大概花了半个小时分析完毕。剩下的都是100万条记录以上的大表了，大概有30多张，我们怕表分析会引起执行计划改变，所以决定在6：00开始做。到6：00时，老肖通过4个窗口，开始分批分析这些表，到晚上7：30，除了半夜要做分区的几张大表，所有的分析操作全部完成。这部分工作顺利完成，大大减轻了半夜时的操作压力。

晚上7：30，我们几个和张工他们一起开了个短会，开发商的小孙也参加了这个准备会。准备会的内容很简单，就是明确各自的工作任务以及接口，并且明确如果优化失败，最后回退的时间定在凌晨4：00。如果本次优化存在问题，在4：00之前一定要停止，并且进行回退操作。

晚上7：50，和老肖最后一次确认了所有的实施脚本。老肖开始进行停机前的操作，开始备份参数文件。优化的预案做得十分完整，每项工作都写成了脚本，并且针对每项工作都准备了应急预案。每个时间点完成什么工作，都在一张检查表上写得清清楚楚。检查表都打印了出来，我们几个人手一份，每完成一个工作，每个人都必须在检查表上打钩签字。最后这份检查表将由客户存档。为了保险起见，我们规定所有的脚本都不允许直接运行，而是通过命令复制粘贴的方式运行；在每条命令提交前，都需要有人负责复核。今天的操作员是老肖，老于负责复核，我作为后备，主要协调小孙测试应用。

8：00时，张工和监控室确定停机时间后，对大家说："可以开始了，哥几个，拜托了！"

8：00时，小孙开始了第一项工作，关闭应用系统。这项工作十分复杂，20分钟后，一部分

Tuxedo 的会话还是没有正常关闭。小孙确认这是正常现象，每次关闭应用时都会有部分 Tuxedo 的会话无法正常退出。我们为关闭应用预留的时间是 30 分钟。为了确保进度不受影响，大家决定直接杀进程，小孙手头有现成的脚本，执行后，所有的会话全部退出。

在小孙停应用的同时，老肖已经修改完了参数文件。修改参数文件采用的是通过 SPFILE 生成 PFILE，然后直接修改 PFILE 的方式。修改完毕后，直接通过 PFILE 生成 SPFILE。

8：30，应用系统正常关闭，数据库很快也关闭了。8：35，监听进程也停止了。下一步是调整操作系统参数。操作系统参数修改早已写成了 shell 脚本，老肖直接将 shell 脚本中的命令复制出来，在 shell 下执行。脚本执行完毕后，老于通过另外一台终端，使用预先准备好的校验脚本检查了操作系统参数修改的结果，确认无误后，大家在检查表上签字确认。

8：50，Oracle 参数修改完毕，数据库顺利重启。下一步是将部分 PL/SQL 对象缓存到共享池中。

9：05，扩充 temp 表空间完成，开始实施表分区操作。在做表分区的同时，日志文件开始调整。日志文件调整采取的方法是先删除老日志组，然后创建新日志组。原计划是分区表调整完毕后才实施，为了节约时间，日志文件调整和分区表同步进行。

2：30：分区表重组完毕，今晚的调整工作基本完成，下面只剩下表分析工作了。重启了 Listener，小孙开始重启应用。不到 3：00，应用重启完毕。下一步就是校验应用了。到目前为止，一切都很顺利，只要确认无误后，就可以完成本次实施了。大家都坐在一边，等着最后的结果。

应用服务器无错误信息，Pass；

数据库警告日志正常，Pass；

应用软件登录正常，Pass；

应用软件日志无报错信息，Pass。

好像一切都很顺利，应用系统正常启动了。大家都十分兴奋，看样子今天的工作可以顺利结束了。

“有问题，这个界面显示不出来”，小孙突然叫了起来，气氛一下子紧张了。一个日常扎账查询模块界面显示不正常。从应用日志上看不到任何异常，WebLogic 的日志上也看不到错误信息。问题十分严重，目前已经快 4：00 了，我们的停机窗口到 8：00 结束。如果在一个小时内无法解决问题，可能就要实施应急预案了。到底是哪个环节出了问题呢？

从数据库警告日志上没有看到任何疑点，也不存在无效的 PL/SQL 对象，索引都是正常的。好像一切都很正常。我和老于都感觉可能是参数调整有些问题，那到底是哪个参数出了问题呢？既然日志没有任何有参考意义的信息，那么唯一的办法就是跟踪应用了。而现场的软件维护人员根本不知道这个模块调用了哪些 SQL，应用软件也没有 Trace 的开关，唯一的办法就是使用 10046 来跟踪 SQL 了。我和老肖说：“小孙，马上关闭应用，把连接池设置为最小，然后把整个数据库打开 10046 Trace，看看到底有什么问题。”

打开 10046 Trace 后，问题很快就找到了，有一个访问 dblink 的 SQL 报 ORA-24370 错误。通过分析，发现这是由于一个 JDBC 引擎的 bug 引起的。在一个通过 dblink 访问的 SQL 里，如

果使用了绑定变量，那么执行 SQL 时会报 ORA-24370 错误。实际上，这个 SQL 没有使用绑定变量，而将 cursor_sharing 设置为 similar 后，该变量被转换为绑定变量，所以执行时就会报错。看来 cursor_sharing 只能回退为 exact。将 cursor_sharing 参数回退后，故障果然消失了。

凌晨 4：50，应用测试终于完成。除了快结束时的小插曲，基本上一切正常。大家都松了一口气。张工看样子还没有从刚才紧张的气氛中缓过劲来，他对我说："老白，你们以前搞优化经常碰到这种情况吗？"我说："碰到这种情况很正常，因为大多数情况下都没办法在优化实施前做严格的应用测试，有些优化操作本身就是有风险的。我们能做的有两点：一是在制订方案时尽可能减少这种风险；二是一旦出现异常，能够尽快解决，就像今天晚上这样。今天这个问题，我们已经考虑了 Oracle 的版本，Oracle 9.2.0.5 是不会有这个问题的，但是我们没想到的是，你们的应用服务器用了一个 Oracle 8.1.7 的 JDBC 引擎来访问数据库，而没有使用 Oracle 9.2 的 JDBC 引擎。下一次我们要更小心一点才是。"张工说："是啊，今天真是有惊无险，不过也够悬的，幸亏问题定位得快，否则明天就不好交代了。对了，你们碰到过回退的情况吗？"

我回答说："我做优化倒是没碰到过回退的情况，不过我看到过。大概是 1 年前了，我在一个联通的客户那边搞调优。正好那天晚上碰到客服系统升级，升级完成后，进行压力测试时，发现性能出现问题，要紧急回退。那个系统是 C/S 结构的，老系统必须在 Windows 98 上跑，新系统必须使用 Windows 2000，所以当天晚上把所有的 Windows 98 系统全部格式化掉重装了 Windows 2000。要回退必须重新将所有的客户端都安装上 Windows 98，当时全省有上百台微机需要回退。后来动员了所有懂点电脑的人都来安装操作系统，才在第二天营业前完成了回退，那次才叫惊心动魄。连我都被动员去安装了两套 Windows 98。"听到这个故事，张工大笑了起来："幸亏我们的系统是三层结构的。"

和张工他们分手，回到酒店已经快 6：00 了。明天，哦已经是今天了，我还要去营口办点事。看样子只能在长途车上睡觉了。

今日点评

优化实施是十分消耗体力的。不管前面的准备工作做得如何充分，在具体实施时还是经常会碰到一些问题的。因此在实施团队中，应该做好分工，每个人的体力也要合理支配。比如，老白在这次优化中担任的是后备的角色，这个角色虽然不是实施的主力，但却十分重要，因此也往往会由经验丰富的老手来担任。这个角色的主要职责是出现问题时出面解决。因此在优化实施之初，担任这个角色的人一定要充分休息，没事的时候不要参与到实际工作中去。既然有了分工，就要充分信任自己团队的其他人。如果你的定力足够，那么别人实施时你就找个地方睡觉去吧。一般来说，这种需要超过 10 个小时的实施项目，到最后的时候，往往是体力出现了问题，导致出现昏招，所以参与实施的每个人都要在前一天睡一个好觉。

在这个项目中，老肖负责实施，老于负责核查监督。他们两个在实施时需要互相协助互相提

醒，特别是在优化实施的后半段，人虽然比较亢奋，感觉不到困乏，不过那时候人的体力已经出现了问题，也是最容易出现问题的时候。可能有人问了，为什么不能把所有的过程做成自动化脚本，直接跑脚本就行了。实际上，在优化实施过程中，经常会碰到一些问题。如果一个脚本要跑上几个小时，那么这个脚本跑下去，一旦出现了什么问题，那么发现的时候就晚了。脚本写得再好也不如人的眼睛，所以老白做实施时一般还是十分谨慎的。要知道我们面对的系统往往都是十分重要的系统，一旦第二天无法正常营业，那问题就严重了。

优化小技巧 升级维护窗口的确定

我经常看到一些 DBA 的实施方案写得相当好，也十分详细，不过这些方案里面对升级维护窗口的设计往往存在一些问题。在一个升级优化工作中，如何设计维护窗口呢？有经验的 DBA 往往会尽可能多地申请维护窗口。如果一件事情要做 2 个小时，有经验的 DBA 巴不得申请 8 个小时的维护窗口，而一个没有经验的 DBA 往往会把自己的停机窗口设计得过小。

一般来说，设计停机窗口时，一些比较明确的操作是可以精确确定操作时间的，这方面不需要做冗余考虑。对于一些没有经过严格测试的操作步骤，一般来说需要根据以往的经验或者测试的结果，在确定这种操作时间时需要设计一些冗余，根据实际情况冗余可以设计为 20%～50%。在总的维护窗口统计出来后，还需要增加一些冗余时间，以防在实际操作过程中出现意外。

在设计维护窗口时，也要充分考虑到回退所需要的时间。维护窗口不能挤占了回退操作的时间，无论停机窗口再紧张，也要预留出足够的回退时间。否则一旦出现问题，就没有足够的时间回退了。实际工作中做回退操作的机会十分低，不过小概率事件不等于不发生的事件，回退时间是在最坏的情况下保命的措施，因此一定要充分预留。

优化小技巧 回退预案的作用

在今天的升级过程中使用了回退操作。不过由于事先没有做好充分的准备，回退是在无预案的情况下做的，因此整个过程显得有点惊心动魄。最后还是在几个老鸟丰富的经验下，很快定位了问题。通过回退 `cursor_sharing` 参数，解决了问题。如果这件事让一个经验不是很丰富的 DBA 去做，那么今天的这个升级只能以失败告终。

在这件事上，可以看到回退预案的重要性。我参与过很多大大小小的升级或者优化项目，这些项目的实施方案都十分周密，不过总的来说，回退预案这方面都十分欠缺，甚至绝大多数项目中根本没有考虑回退预案。还有一些只是简单的几句话：实施前备份系统，如果出问题就使用备份系统恢复。这种预案几乎是没用的，因为在实施过程中，不仅仅是对文件和数据进行修改，回退也不是回退到初始的状态，而绝大多数情况下需要做的回退操作是局部的回退，因此针对可能发生的情况需要设计响应的预案。

比如，我们这次操作需要涉及 `cursor_sharing` 方面的操作，那么和 `cursor_sharing` 相关的一些预案就要齐备，包括可能出现的一些问题以及解决方案。预案不可能十分完备，也不可

能预见到所有可能存在的问题，不过多一份预案，对你的工程就是一份保障。

我参加过不少系统的上线护航工作。在参与项目之前，我都会了解本次护航涉及哪些系统、和哪些技术相关。我会准备好所需要的工具以及一些故障诊断工具和脚本，一旦出现问题，就可以马上使用。2008 年 11 月，我参与中国最大的一次运营商系统割接工作。在前一天，我就和集成商的 DBA 一起准备监控环境，因为我不是铁人，不可能连续三四天 24 小时都盯着系统。因此，我们准备好了针对不同数据库的监控手段，针对 Oracle 9i 数据库，我们安装了一台 OMS 服务器，并且通过监控录像的方式对其主要的性能指标进行监控。针对 Oracle 10g 的数据库，由于有 ASH 和 AWR，所以省了不少心。一旦出现故障，可以通过 ASH 立即进行分析。如果我们不在现场，也可以通过 AWR 等报告来回顾故障发生时的情况。

不过虽然做足了功课，还是有意外的事情发生。我们负责监控的割接相关的数据库均没有出现明显的问题，倒是本地网的一个 RAC 数据库的一个实例突然宕了，还好我们第一时间发现了这个问题，并且及时重启了数据库。通过检索资料库，很快定位了故障的原因。由于处理及时，定位问题准确，甲方对此也十分满意。快速定位问题的诀窍就是我自己的知识库，以前我在网上收集到好的技术资料时，都会存放在某个文件夹里，时间长了，这个知识库就很难检索了。后来发现 Firefox 下的 scrapbook 这个插件可以十分方便地将网页拖拉到特定的知识库里面，并且这个知识库可以创建全文检索索引。通过全文检索索引就可以很容易地进行关键字检索。这个知识库我已经积累了一年多了，目前已经积累的文档总数超过 4 000 个，日常维护工作中发生的主要问题都可以在这个知识库中搜索到。

6月7日 突发事件

昨天晚上一夜没睡，经历了凌晨那次小的惊吓后，大家都有点后怕。回酒店的路上，老于和我都还暗自庆幸。回到酒店后已经 6 点多了，如果这个时候睡觉，等会起来会很痛苦，所以我只是稍微躺了一下，洗了个澡。看了一会儿“第一时间”，就已经 8 点多了。我最好能够赶早班车去营口，才不会影响今天的工作。简单拿了两件衣服，塞在电脑包里，就出门了。

刚下楼就看到老于他们三个人在楼下拦出租车。我问老于：“上午老熊不是值班吗，大家怎么不休息休息”。老于说：“不太放心，还是过去看看，下午再睡觉”。于是我就蹭老于他们的车一起去北站旁边的客运总站。我的运气还不错，刚到客运站就赶上有辆车要走，连忙和司机打了招呼就去买票。

车刚开出没多久，我抱着电脑包就昏昏沉沉地睡着了。自从上次在厦门电脑包被掉包后，我就不敢把电脑包放在行李架上，每次坐长途车都抱着电脑包。车刚刚上沈大高速，老于的电话就打进来了。一看来电是老于的号码，我就感觉可能是系统出了问题。

现在的时间是早上 8：40，今天是周末，按理说系统的负载应该不大，但是从 8：30 开始，系统的 CPU 突然高达 100%，I/O 也很大，连在操作系统上执行 ps 命令都感觉有点慢。优化后不但没有看到性能提升，而且出现了严重的性能问题。老于他们观察了一下，有一张我们昨天晚上做了分区的大表出现了大量的全表扫描。老于和我都怀疑是不是分区导致了全表扫描呢？我们在做表分区的同时，所有的索引也都使用了本地索引，按理说不应该出现以前走索引的操作改走全表扫描的问题。

我仔细想了一下，感觉按照我们的方案不应该会出现执行计划的突变，除非是昨天重建完表后没有做分析。昨天我们走时表分析并没有结束，提交了后台作业后就离开了，会不会是分析出现了错误呢？我建议老于他们检查一下分析数据。老于说他们开始也怀疑分析没有正常完成，老熊刚刚检查过，分析数据正常，并且采样比例确实是 30%，应该没问题。于是我建议他们马上检查一下是否存在索引失效，或者某个分区索引有问题。老于说他们已经检查过了，那张表上的 6 个索引全部都是正常的，而且分析数据也很正常，按照 30%比例采样的，采样比例也应该足够。

我马上打开电脑，由于长期从事维护工作，所以一般来说在旅途中，我的电脑都处于待机状态，以便尽快处理突发事件。打开电脑后，首先我检查了保存在电脑上的 dba_indexes 视图的信息。在做优化项目时，我有一个习惯，就是会保存每个阶段的主要系统视图。这个习惯被证明

是十分有效的，已经多次帮助我发现问题。

打开 Excel 后，我立即发现有些不对。因为在我的记录里，这张表有 7 个索引，而刚才老于很明确地说 6 个索引全部正常。于是我立即打电话询问老于，那张表上到底有几个索引，到底是 6 个还是 7 个。老于确认说只有 6 个索引。通过核对，我发现在目前系统里，比 5 月 13 日采集数据时少了一个索引。

老于他们使用 OEM Top SQL 工具抓取了相关 SQL，通过比较 SQL 的 `WHERE` 条件，可以确定就是由于这个索引缺失引起的。由于这个索引的缺失，造成了一张千万条记录的表的访问产生了大量的全表扫描，从而导致 CPU 出现了严重的问题。按照我提供的索引字段添加遗失的索引后，CPU 使用率骤然下降，稳定在 30%左右。由于今天是周末，所以无法推断在业务高峰期，系统性能是否有所改善。不过通过对一些常见 SQL 的分析来看，无论是物理读还是逻辑读，平均每次执行的开销都有明显的下降。如果不考虑排队效应，系统整体负载下降和性能提升是肯定的。但是生产环境改善效果的评估还需要经过下周的观察才能最终确定。

问题是解决了，但是为什么会遗失一个索引，这个确实十分费解。老肖是经验很丰富的 DBA，做表重建方法也不会存在丢失索引的可能。表重建的方法，首先通过工具取出建表的相关脚本（可以使用 Toad 或者类似的工具，在这个项目中我们使用了 Toad），然后将表重命名（包含所有索引），然后重建表，再将数据用 `INSERT /*+ APPEND */ SELECT...`的方法从原表中导入数据。这种过程不可能出现索引丢失。旧表改名后还没有删除，因此老于他们比对了改名后的表，发现也只有 6 个索引，而不是我采集的 7 个索引。唯一的可能是我采集了 `dba_indexes` 信息后，有人删除了某个索引。

如果真有人在我们实施前删除了一个索引，删除索引的目的是什么呢？是误操作还是有什么其他意图呢？看上去在我们实施前有人删除索引是唯一的可能了，但是处于这种多方参与的项目中，团结是第一位的，因此我们决定这个问题到此为止。如果客户要追究责任，那么我们承担下来就可以了。这个问题就不再升级了，否则追究起来也不可能有什么结果，反而会在几个团队之间形成不友好的气氛。我们决定只是通报一下张工，由于某个 SQL 造成了大量的全表扫描，影响了性能，我们紧急处理，添加了一个索引，问题就解决了。其他问题就不再提了，反正问题发现得早，没有造成太大影响，张工那边也好交代。看来今天老于他们提前去现场还是正确的，否则这个问题造成的影响大了，就不好处理了。

下午张工他们到现场转了一圈，系统一切正常，下面各个营业厅也没有相关的投诉，看样子我们担心的负面影响是不存在的，这次优化基本上成功了。张工他们来的时候，老于还是按照我们商定的方案，主动把添加索引的问题和张工说了，由于发现及时，处理得当，故障并没有造成明显的损失，所以张工也建议这个问题不要升级了，只是下次优化操作时要注意不要发生类似的情况。

晚上和客户一起吃完饭，我还是给老于打了个电话，问问现场的情况。老于下午对一些主要的 SQL 都进行了采样。通过采样比对，这些 SQL 的平均开销和平均执行时间都有了较大的改善。不过有一些 Top SQL 由于周末的原因，并没有执行，不过老于他们还是对部分执行次数较多的

SQL 进行了手工试运行，运行的结果也颇令人放心，这些 SQL 或多或少都有所改善，I/O 和缓冲区获取的开销均有一定幅度的下降。看样子周一系统应该会出现较为明显的改善迹象。

今日点评

今天老白的好习惯又一次发挥了作用。每次做优化时，老白都会对主要的数据字典进行备份，并且用 Toad 将这些字典表的数据用 Excel 格式的文件保存起来，以便于今后出问题时核对。今天出现的问题正好用上了这些 Excel 文件。通过这些文件，很快就定位到丢失了一个索引。

其实需要留意的字典表主要涉及表、索引、别名、视图、存储过程、触发器等方面。

优化小技巧 使用 HINT 优化 SQL

HINT 是一种 SQL 优化的重要手段。在默认情况下，SQL 的执行计划和优化方案是数据库的优化器自动完成的，不需要编程人员干预。但是数据库优化器的优化方案和执行计划不总是最优的。Oracle 数据库提供了开发人员强制性制订优化方案的方法来弥补数据库优化器在这方面的不足，这种方法就是使用 HINT。

HINT 的使用方法是在 SQL 语句中插入/*+ <规则> */ 这样的提示，这些提示需要放在 SELECT、UPDATE、DELETE 这些词的后面，比如：

```
SELECT /*+ FIRST_ROWS */ * FROM T_TEST WHERE R>10;
```

要注意的是，/*+一定要连写，之间不能有空格，后面一定要有空格（在某些早期版本中，缺少空格可能会导致 HINT 失效）。另外，需要注意以下几个问题。

- 提示将使语句强制执行基于成本的优化器（除了提示规则外）。
- 如果表名包含类似<schema>.<table>或<table>@<db_link>这样的格式时，在 HINT 里要使用表的别名。
- 确认相关的表都分析过，并且最近一次分析能够反映出该表现在的状态。

Oracle 的优化器各有特点，RULE 优化器是完全基于规则的，在今后的 Oracle 版本中将逐渐被淘汰。FIRST_ROWS 优化模式强调的是以最快的速度返回第一批数据，而不考虑综合的 I/O 开销。ALL_ROWS 优化器考虑总体 I/O 最小，而不考虑响应速度。CHOOSE 优化器是在几种优化器中选择认为是最优的方案。

优化器可以在数据库级进行定义（设置参数），也可以在会话级定义。通过下列语句，可以修改会话的默认优化器：

```
ALTER SESSION SET optimizer_goal= rule | first_rows | all_rows | choose ;
```

而使用这两种方法修改优化器模式都存在一定的局限性。对于不同的 SQL 语句，可能需要使用不同的优化器来优化。我们可以通过优化器 HINT 来规定某个语句使用某种优化器来优化。

优化模式如下表所示。

优化模式	说　明
FIRST_ROWS，ALL_ROWS	强制使用CBO FIRST_ROWS、ALL_ROWS优化器
RULE	尽可能使用基于规则的优化器
ORDERED	访问表的顺序按照FROM语句中的顺序
ORDERED_PREDICATES	使用WHERE语句中的顺序来确定执行计划，不通过索引键值来评估

视图或子查询都涉及多表连接。多表连接和子查询的执行计划有多种，可以根据语句的需要来使用这些 HINT 优化独立的查询。

HINT	说　明
PUSH_SUBQ	让子查询尽可能早地执行。通常状态下（Oracle 7.2及更高版本），子查询都在比较靠后时被执行
NO_MERGE(v)	如果在一个视图上设置该HINT，禁止子查询和父查询合并。如果在使用该视图的父查询中使用该提示，禁止该视图被合并（Oracle 7.2及更高版本）
MERGE(v)	合并视图
MERGE_AJ(v)	放在一个NOT IN子查询中，把一个子查询转为MERGE SORT(ANTI)模式去进行表连接操作（Oracle 7.3及更高版本）
HASH_AJ(v)	类似上面，只是使用Hash连接。 例如：SELECT .. WHERE deptno is not null AND deptno NOT IN(SELECT /*+ HASH_AJ */ deptno ...)
HASH_SJ(v)	把EXISTS子查询转为散列
MERGE_SJ(v)	把EXISTS子查询转为合并
PUSH_JOIN_PRED(v)	把连接模式放入视图中
NO_PUSH_JOIN_PRED(v)	不把连接模式放入视图中

通过 HINT 我们还可以定义数据的访问方式，比如通过索引访问、全表扫描等。

HINT	说　明
FULL(tab)	使用全表扫描
CACHE(tab)	如果表大小小于cache_size_threshold，把该表视为设置了CACHE选项，仅仅当使用FTS时才有效
NOCACHE(tab)	不使用CACHE选项
ROWID(tab)	直接使用ROWID访问SELECT /*+ ROWID(table) */ ... FROM tab WHERE ROWID between '&1' and '&2';
CLUSTER(tab)	使用CLUSTER扫描
HASH(tab)	使用HASH扫描
INDEX(tab index)	指定索引
INDEX_ASC(tab index)	指定索引进行范围扫描
INDEX_DESC(tab index)	使用降序索引进行范围扫描
INDEX_FFS(tab index)	使用全索引快速扫描取代FTS
INDEX_COMBINE(tab i1.. i5)	使用多个位图索引的结合来访问某个表
AND_EQUAL(tab i1.. i5)	取多个独立索引的交集，可以取代复合索引
USE_CONCAT	使用UNION ALL替代OR或者IN
NO_EXPAND	不要使用USE_CONCAT
DRIVING_SITE(table)	强制查询在表所在的节点进行

表连接有很多种方式，不过 Oracle 的优化器很可能会选择错误的表连接方式。如果碰到这种情况，我们可以通过下面的提示来进行优化。

HINT	说　　明
USE_NL(tab)	把tab作为嵌套循环的驱动表，只有在使用ORDERED时才有用
USE_MERGE(tab..)	在一个排序-合并连接中使用tab作为驱动表，只有使用ORDERED时才有用
USE_HASH(tab1 tab2)	使用Hash连接tab1和tab2
STAR	如果可能，使用星型连接
STAR_TRANSFORMATION	如果存在星型连接，选择一个最佳的连接方式

并行方面的操作也可以通过下列提示来指定。

HINT	说　　明
PARALLEL(table, <degree> [, <instances>])	使用并行访问某个表
PARALLEL_INDEX(table, [index, [degree [,instances]]])	在分区索引上使用并行范围扫描
PQ_DISTRIBUTE(tab, out, in)	选择在一个并行查询中分发某个表的方法（out和in的取值范围：hash/none/broadcast/partition）
NOPARALLEL(table)	在某个表扫描时不使用并行
NOPARALLEL_INDEX(table [,index])	在索引扫描时不使用并行

除了上述提示外，还有一些提示也是十分有用的。

HINT	说　　明
APPEND	在INSERT...SELECT时有效，允许INSERT使用类似DIRECT LOAD的机制
NOAPPEND	不使用APPEND
REWRITE(v1[,v2])	Oracle 8.1以后新增功能，如果MV在列表中就使用，不在列表中就使用符合条件的MV
NOREWRITE	Oracle 8.1新增功能，不重写查询

比如 APPEND 是我们在 INSERT...SELECT 语句中常用的提示。使用 APPEND 提示可以在高水位以上的部分插入数据，从而减少 INSERT 语句的成本，大大提升 SQL 的效率。

6月10日 使用自动采样比例引发的故障

由于营口的事情比较复杂，原本准备周一回沈阳，只能推迟一天了。周一的时候，听老于说系统情况很好，调整过的系统都有较明显的改善。CPU 下降了 15%～20%，I/O 也有明显的改善，所有指标都处于正常范围，内存还剩余 1GB 左右，这些改善都为下一步优化提供了很好的基础。虽然我对这个效果早有预期，但是听到这个消息仍然很高兴。由于沈阳那边都很顺利，我决定在营口多停留一天，所以周二在长途大巴上时，我的心情还是挺放松的。一般来说，系统从 8：30 开始压力加大，9：30 左右是一天中最高峰的时段，直到 11：30 午休时压力会下降一些，下午 3：00～4：00 又是一个业务高峰，这个高峰持续到 5 点多逐渐减弱，不过 5：40 至 6：00 之间由于营业扎账，客服系统和收费系统会出现一个短暂的压力高峰。这种现象被做优化的人称为系统的节奏。掌握系统节奏在优化分析中十分关键。

大巴开车时是早上 8：00，现在已经接近 9：00 了，还没有接到“骚扰”电话，说明一切正常。经过周一的考验后，我对优化的效果还是有信心的。回到沈阳已经快 11：00 了，我一进办公室就感觉气氛十分紧张。老于和老熊正在紧张地敲着键盘，老肖昨天已经回北京了，目前沈阳只有老于和老熊值守。

看到我进来，老于马上跑过来说：“老白，我昨天晚上干了一件蠢事，周一下午和开发商讨论定期分析表和索引时，开发商认为分析一遍所有的表时间太长，能不能让分析速度加快一点。我找了一下 Oracle 9i 的相关资料，发现 dbms_stats.gather_table_stats 的采样比例参数有一个值，就是自动采样比例。我就考虑用 Oracle 9i 的自动采样比例。我在自己的数据库上做了个测试，发现使用自动采样比例分析很快。想今天验证一下自动采样比例是否可行，就对一张有 2 000 万条记录的表做了一个实验，今天这张表相关的业务就出问题了。今天早上 10：30 开始，有一个 SQL 导致这张表的全表扫描，CPU 产生了严重的瓶颈，现在系统基本上处于不可用状态。”

开发商的维护人员小孙已经通过电话咨询开发人员，确认了这个 SQL 是由一个叫“机打发票”的程序产生的，这个应用每个月只有出账后的几天使用，一般就是 8 日、9 日，这个程序主要用于批量打发票。由于我们是 11 日才进场的，所以我们没有采样到这个 SQL。这个 SQL 正好使用了昨天老于做自动采样分析的那张大表，老熊也确定是由于执行计划不正确导致的问题，目前这张表的采样比例是 5%。老熊建议立即对这张表做重新分析，不过采样比例他没有把握，建议 30%。我考虑了一下，建议马上使用 20%比例采样，分析一下表，暂时就不要带 cascade 参数了，索引今天晚上找时间再分析。

老熊的脚本都已经准备好了，听了我的建议后，马上调整了参数，开始对表进行分析。老熊刚分析完这张表，正准备测试一下那个 SQL 的执行计划是否改善了，突然发现数据库不可访问了。我们正准备查查数据库是否宕了，服务器也连不上了。开发商的小孙告诉我们，维护组那边决定重启服务器了。20 分钟后，服务器重启完成，应用也重新装载了。系统重启以后，一切恢复了正常，查看那个 SQL，执行计划完全正确，全表扫描消失了，CPU 使用率又恢复到 60%～70%，I/O 情况也十分正常，WIO 在 10%左右。

大家刚刚松了一口气，孙主任和张工他们急匆匆地走了进来，说："老于，老白，我们开个小会，把刚才的问题分析一下。"

原来今天的问题影响很大，连孙主任也被从总经理办公会议的会场找回办公室，集成商那边分析认为有一张近 100 万记录的表上少了一个索引，导致了今天的性能问题，所以他们新建了一个索引，并且重启了服务器，问题就解决了。孙主任介绍完情况后又问我："老白，你以前提出过这个服务器只有 2 颗 CPU，已经很难承担目前的业务量了，建议我们扩容。这个扩容是不是特别紧急？如果必须扩容，那么我们可以加快采购的进度。采购 CPU 的事情我们已经在走流程了，不过到货还要半个多月，如果实在来不及，我们也可以考虑买水货，尽快到货，你们作为优化厂商，有什么问题，不要看我们这边的眼色，该怎么样就怎么样。开发商认为 CPU 已经承受不了了，他们多次提出扩容，只是你们前一阵子说通过 6 月 6 日的优化可以解决 CPU 的问题，我们才没有对 CPU 采购加急。"

看得出来，孙主任是真急了，他也认为今天的问题是由于 CPU 瓶颈造成的。于是我把刚才问题的原因原原本本地和孙主任汇报了，老于也向孙主任表示抱歉。我说，问题的关键不在于开发商加的那个索引，而是由于表分析采样比例不足导致 SQL 执行计划的改变，因为这个业务昨天就有人在做，而昨天系统都很正常，昨天晚上老于重新分析了这张表，今天才出现问题的。系统扩容虽然是必需的，但是由于我们优化的效果还不错，一个月之内，系统不会出现明显的瓶颈，半个多月的采购周期不会有太大的影响，他们可以按照原来的采购计划进行。

最后，我说："孙主任，今天的问题，其实是我们的失误造成的，不过也是我们解决的，重启服务器和添加索引并不能真正解决问题。"

孙主任说："不管怎么样，开发商认为他们加了索引重启了服务器就解决问题了，所以你们也不要过于矫情了，这个问题就算是人家解决的吧。明天例会时，也不要提谁解决的了，关键是这种事情不能再发生了。"

中午吃午饭时，老于还在讨论自动采样比例的事情，老于说："我搞了 Oracle 差不多 10 年，大多数时间玩的都是 Oracle 8i，Oracle 9i 的新技术用的不是很多，而且我从来对新技术都是小心翼翼的，这回偶尔玩了一下新技术，就玩出了毛病，下回更不敢随便玩了。"

是啊，新技术是好东西，但是又有多少人在用之前弄明白了呢？连老于这么谨慎的人都会禁不住诱惑去使用自动采样比例，这是 Oracle 的官方文档对自动采样比例的吹嘘太夸大其词了。实际上，采用自动采样比例出现问题的案例比比皆是，如果不经过严格的测试，绝对不能随便使用。老熊在旁边补充说："不管怎么样，我做表分析，采样比例一般尽量选择高一些，小表 40%以上，

大表 30%以上，这样出问题的可能性比较小。”

对于表分析的采样比例，确实没有一个很严格的标准，不过出于安全考虑，对于没有经过测试的系统，最好不要使用较低的采样比例，老熊这种做法是比较安全的。一般来说，采样比例低于 10%是比较危险的。不过有些系统由于数据量很大，采用较大的采样比例不太现实，这种情况下就需要做足测试。如果试验证明 5%的采样比例是安全的，那么也确实没有必要去使用 30%的采样比例了。

今日点评

今天是由于老于尝试使用自动采样比例导致了一张重要的表采样比例过低，大量 SQL 的执行计划出现了问题，从而导致系统性能问题。老白经常强调 DBA 不能太好奇。实际上，今天这种问题完全是可以避免的。原本第一次优化的效果十分不错，没必要再去画蛇添足了，但是有时候人还是很脆弱的，经常会做一些莫名其妙的事情。老于事后也说不知道为什么前天晚上他就做了这件事，导致了这种后果。老白也曾经碰到过一件事情，差点导致严重的后果。当时在一个客户那里帮助客户做一个健康性检查，所有的活都干完了，报告也写完了。按理说这个时候我和客户提出今天的活干完了，客户也不会有什么意见。不过上午时听客户说他们的系统经常会出 ORA-4031，健康检查时也确实看到了几份 ORA-4031 的日志，所以突然想帮客户分析一下共享池。我手头的一套分析共享池的脚本，在 Oracle 8、Oracle 9、Oracle 10.2 上都用过，没什么问题，因此也很自信地跑了一下。当时的环境是 Oracle 10.1.0.2，脚本跑了一分多钟，突然客户跑过来说系统挂了，让我帮助看看。我立马看了看我跑的脚本，发现是我的脚本导致了系统挂起，虽然两三分钟内我就杀掉了进程，解决了这个问题，不过后来想想确实是挺悬的。虽然你帮客户做一些额外的工作是出于好意，不过你的好意可能会带来负面效果。有时候风险的评估往往会超出我们知识的范围，这个时候就要十分小心了。

优化小技巧　关于表分析比例的判定

经常有客户问我表分析比例如何判定。实际上，针对不同的应用系统，确实没有一个标准答案。对于不同的应用系统，都需要做严格的验证。实际上表分析采样比例主要取决于数据分布的均匀程度。对于十分均匀的表，采样比例小一点，都可以获得较为准确的结果，而对于数据分布不均匀或者某个字段的值存在严重倾斜的情况，采样比例就要高一点。

对于一些小系统来说，在维护窗口中做一次 100%的分析是没有任何问题的，因此这类系统可以采用 100%采样。而对于一些海量系统来说，能够采用较低的采样比例来确保执行计划不出问题，那么尽可能地使用较低一点的采样比例。一般来说，每张表的最低采样比例的测试，最好在系统上线前或者上线后不久就完成，今后只需对某些表进行微调。

而实际工作中，我们往往没有那么幸运，因为我们面对的系统往往都是已经运行了数年的老系统。在这种情况下，如果有个客户由于数据量增长，无法保证 100%采样，希望你帮他出出主

意，给出一个最低采样比例的建议，那么你将面临一个巨大的挑战。因为你面对的可能是一个十分关键的业务系统，一旦出现问题，将万劫不复。如果遇到这样的挑战，你将如何面对呢？

想要回答这个问题确实比较难。我们不能拍着脑袋说，采样比例 30%合适还是 50%合适。最好的办法是在生产系统上进行测试，比如先测试 20%，如果系统中主要 SQL 的执行计划都正确，那么说明这个采样比例是适合的，否则就增加采样比例再进行测试。这个方法听起来不错，不过有经验的老 DBA 早就嗤之以鼻了。核心的生产系统是不能让你这么折腾的，而且如何判断“主要 SQL 的执行计划没有改变”也不是一件容易的事情。

既然无法在生产系统中做这个测试，那么就必须在测试环境中做了。我们对测试环境有什么要求呢？通过一个逻辑备份，将数据导入到测试库中是否可行呢？答案是否定的。逻辑备份恢复的数据，在数据块中的数据分布方面已经进行了重组，无法充分反映出生产库中的数据分布情况，因此逻辑备份恢复的数据可以反映出一部分生产库的情况，而不是全貌。我们必须在物理结构完全一致的数据上做测试，才能得到最为准确的结果。在物理 standby 数据库上进行测试是最佳的选择。如果没有物理 standby 数据库，使用 rman 备份的备份集恢复一套数据库，也是不错的选择。

一旦你拥有了一套测试环境，就可以进行所需要的测试了。除了使用 SQL 直接验证采样比例的影响外，直接比对采样结果也是很常用的方法。如果使用不同的采样比例得到的采样结果类似，那么选择较低采样比例的风险也相对较小。比对采样比例时还要注意，不仅仅要比对 dba_tables、dba_indexes 中的结果，还要比对 dba_tab_columns 中主要字段的统计数据，因为这些数据对于执行计划中计算成本影响极大。

优化小技巧　AIX 下使用 **vmstat** 分析 I/O 问题的小技巧

一般 DBA 喜欢使用 vmstat 命令来查看系统的状态。如果问起 vmstat 中 b 的含义，可能大家都很清楚。b 表示资源等待队列的长度。从某种程度上来说，b 等待可以和 I/O 等待等挂钩，如果数据库使用文件系统作为数据文件的存储方式的话。但是对于裸设备来说，裸设备出现等待并不会明显地体现在 b 队列上。

在 AIX 下提供了一个-I 选项。使用该选项，可以看到一个 p 值，p 值代表物理 I/O 线程等待队列的长度。如果 p 队列的长度较大，那说明存储系统的性能存在问题。

```
vmstat -I 1 10

System configuration: lcpu=48 mem=96256MB ent=24.00

  kthr     memory              page              faults              cpu
-------- ----------- ------------------------ ------------ -----------------------
 r  b  p      avm      fre fi fo pi po fr  sr   in     sy    cs us sy id wa    pc    ec
 4  0  2 12901682 9358723  0  0  0  0  0   0 5049 49005 11774 14  1 82  4  4.17  17.4
 1  0  6 12901699 9358706  0  0  0  0  0   0 5232 51973 12165 14  1 81  3  4.35  18.1
 4  0  8 12901708 9358697  0  0  0  0  0   0 4933 48704 12880 12  1 83  3  3.81  15.9
 4  0  6 12901689 9358716  0  0  0  0  0   0 5068 44568 12505 12  1 83  3  3.84  16.0
 3  0  4 12901666 9358739  0  0  0  0  0   0 5936 50968 14031 12  1 84  3  3.76  15.7
 6  0  2 12901687 9358718  0  0  0  0  0   0 6205 54103 14774 12  1 83  3  3.79  15.8
```

```
  2  0  0 12901688 9358717  0  0  0  0  0   0 6541 56430 15516 13  2 83  3  3.99  16.6
  2  0  4 12901678 9358727  0  0  0  0  0   0 6514 54752 14903 13  2 83  3  3.96  16.5
  4  0  0 12901670 9358735  0  0  0  0  0   0 6270 57911 14686 12  1 84  2  3.88  16.2
  5  0  6 12901667 9358738  0  0  0  0  0   0 7369 59685 15826 13  2 83  2  4.17  17.4
```

在这种情况下，可以分析一下 pbuf 是否存在不足的现象。具体方法是使用 vmstat -v，查看是否存在较多的 pbuf 阻塞等待。

```
vmstat -v
             24641536 memory pages
             23657185 lruable pages
              9356511 free pages
                    8 memory pools
              1889295 pinned pages
                 80.0 maxpin percentage
                  5.0 minperm percentage
                 15.0 maxperm percentage
                  9.9 numperm percentage
              2357564 file pages
                  0.0 compressed percentage
                    0 compressed pages
                  9.9 numclient percentage
                 10.0 maxclient percentage
              2357564 client pages
                    0 remote pageouts scheduled
                 2796 pending disk I/Os blocked with no pbuf
                    0 paging space I/Os blocked with no psbuf
                 2228 filesystem I/Os blocked with no fsbuf
                 2315 client filesystem I/Os blocked with no fsbuf
                 5311 external pager filesystem I/Os blocked with no fsbuf
                    0 Virtualized Partition Memory Page Faults
                 0.00 Time resolving virtualized partition memory page faults
```

实际上，小小的 vmstat 中还包含了十分多的门道，用好它可以为我们提供非常多的有效信息。

6月11日 阶段总结

今天本来是例会时间，由于前一段时间刚刚做了优化，需要做一个全面的汇报，小齐也专程从北京过来一起参加，这次会议也改为优化的阶段总结会。虽然优化效果十分明显，第一次调整基本上达到了我们的预期目标，系统性能有了较大的改善，甚至大部分性能指标远远超出了我们的预期，但是由于出现了昨天的系统故障，给客户造成了十分不好的影响，所以大家的情绪都有点低落。

开会前我们几个人单独讨论了一会，大家都觉得如何解释昨天的问题是一个十分关键的问题。如何让客户正确地认识到这次调整的效果，从而对我们后面的工作建立信心也是十分重要的。优化项目实施过程中十分怕出问题，每次出问题，客户对你的信任就会下降，严重时甚至可能出现客户的对立情绪，如果到了这一步，下面的工作就很难开展了。

今天是孙主任主持会议，他首先发言。不过他除了对开发商昨天快速反应最终解决问题提出了表扬外，没有再提到昨天的事故。孙主任中肯地评价了这段时间大家的工作成果，也表示他们对优化项目最终的成功充满了信心，并希望我们在今后的工作中能够更加细致："优化本身就是曲折前进的过程，在这个过程中肯定会出现一些反复，甚至出现严重的倒退，但是只要大家的目标一致，实现目标的手段是正确和合理的，那么最终的结果肯定是好的。对这一点我深信不疑，所以大家不要怕，继续沿着我们规划好的路子走，这个项目成功现在就能看得见，起码我已经看到了希望。"

在这种情况下，作为客户方的领导，能够说出这种话，是我们万万想不到的。大家听了孙主任的发言，心里都有些感动，甚至我的眼眶都有些湿润。我做过几十个类似的项目，但是有这种感动还是第一次。作为项目甲方的第一责任人，昨天我们的失误给他带来的压力是可想而知的。但是这些压力他一个人承担下来了，没有对我们有一丝一毫的抱怨，给与我们的只有鼓励和信任，这一点太难能可贵了。有了这份支持，我们如果还做不好的话，那么就太不应该了。

今天我们也给客户带来了一个惊喜，在会议前我们也讨论了这个问题要不要放到今天的例会上去讨论，后来还是决定在这次会议上把这个惊喜拿出来，也给大家一点信心。这几天老熊已经找到了解决那个12张表连接的大型查询的优化方案了，由于这个SQL里使用了大量的占位操作，导致优化器产生的执行计划不合理，这个 SQL 占用了收费系统 30%以上的系统资源。如果将这些占位操作全部去掉，那么这个 SQL 的执行计划将十分完美，所耗资源不到目前消耗量的 1%。

为了防止开发商以业务太复杂、程序无法实现为理由而不愿意去掉占位操作，老熊还特意设

计了一个存储过程。使用这个存储过程，通过输入不同的条件，可以拼凑出满足各种要求的、不含占位操作的 SQL 语句。实际上，开发商只要根据老熊的存储过程经过小的调整，就可以完成 SQL 拼凑的程序，然后将拼凑出来的没有占位操作的 SQL 提交执行，这样就可以大大减少这个 SQL 的开销。

按照预先设计好的思路，我们先提出了去掉占位操作，可以提高几十倍的性能，从而解决掉收费系统 CPU 瓶颈的问题。老熊展示了使用占位操作和去掉占位操作的 SQL 的执行结果和开销的比较。可以看出结果是相同的，但是开销大大减少了。

开发商很愉快地认可了优化的效果，但是他们马上提出了去掉占位操作的逻辑十分复杂，要重构这个业务模块的逻辑，难度太大，已经超出了他们目前的能力。这种回答是我们预先有准备的，我马上追问："你们的难点在哪里？哪个问题是你们目前无法解决的？"开发商的回答是由于这个模块的业务逻辑十分复杂，迫不得已才使用占位操作符的，如果去掉占位操作，他们将无法根据逻辑条件生成不带占位操作的 SQL。除非我们给他们找到一个解决方案，否则他们很难完成这个修改。

这个问题正中我们的下怀，我们早就知道开发商可能以这个理由来拒绝修改。看到开发商进入了我们的圈套，老熊搞的那个存储过程马上被投影到墙上，通过十分清晰的逻辑，展示了去掉占位操作的 SQL 生成方案。孙主任看后马上说："这个很清晰，连我这个不懂技术的人都看得懂，这么做应该是可行的吧？"

由于开发商没有想到我们居然会准备得这么充分，这种情况下他们也没办法打太极拳了，也只能承认这个方案是可行的。于是我们马上要求他们承诺在 6 月 20 日前完成应用的修改，并在 6 月 25 日前发布补丁包。在我们和甲方的共同努力下，开发商只得承诺在 6 月 25 日前发布补丁包。

今天的例会开得十分成功，出来后，大家都感到放下了一个很大的包袱。中午吃饭时，老于十分感慨："这种用户太仗义了，如果我们不尽心尽力帮他们解决问题，就白活了。"孙主任今天的宽容大度让大家都有一种滴水之恩当涌泉相报的感觉，小齐也说："碰到这种用户，是我们的福气，我们绝对不能辜负他们。"

小齐下午就要回北京，所以我也希望她回北京后能够和集团公司沟通，把今天开发商承诺修改那个应用的结果巩固下来。这个 SQL 如果能够修改，那么收费系统的优化问题基本上就解决了。虽然会上大家都达成了一致，不过我感觉如果不通过高层施压，这个问题没那么容易解决，开发商的惰性比我预想的还要大。由于集团公司那边有下一期合同作为砝码，他们的话开发商还是能够充分考虑的。集团公司出面向开发商的高层施压，甚至比孙主任他们说话分量还要大。

今日点评

如果在优化项目中碰到了类似今天遇到的问题，该怎么办呢？这种情况下，客户最不喜欢听到的是推卸责任的话，因此在汇报时尽可能把责任承担下来，并且诚恳地向客户表态这种问题不会再发生，希望客户给我们更多的机会。如果这个时候去强调客观原因，那是会适得其反的。因

为实际上项目是客户的项目，客户和我们应该是一荣俱荣、一损俱损的，所以客户和我们的目标实际上是相同的。如果这个项目做砸了，那么客户受到的影响比我们还大。所以无论如何，客户都会帮助我们走出困境的，而这个时候如果你还去推卸责任，只能激起客户的怒火，而不可能把你的责任推卸干净。勇于担当也是 DBA 在特殊环境下必须拥有的性格。今天孙主任并没有对我们提出很严厉的批评，这也是一个成熟的甲方的正确选择。因为孙主任是这个项目甲方的负责人，这个项目的成败对他的影响是最大的，而孙主任最终依赖的力量就是我们这批干活的人，所以他不把在公司领导那边承受的压力转嫁到项目实施小组身上是十分明智的。

优化小技巧 STATSPACK 报告在优化项目中的作用

在本书中，很多地方提到 STATSPACK，那么 STATSPACK 到底是一个什么东西，它如何在数据库优化中发挥作用呢？说到 STATSPACK，就不能不说说 OWI（Oracle Wait Interface）。OWI 是 Oracle 的数据库性能分析接口，提供了一系列的视图。通过对这些视图的访问，就可以了解 Oracle 性能状态。这些视图包括以下几种。

- v$system_event：总体性视图。
- v$session_event：按照会话划分的总体性视图。
- v$session_wait：明细信息，每三秒钟刷新一次等待时间。

STATSPACK 作为性能分析工具是基于 OWI 的，它通过采样获得当前系统状态的快照。通过两个采样点，就可以生成出一份 STATSPACK 报告，其中包含了这两个快照点之间的系统的性能状态。通过解读这份报告，可以很详细地了解系统的性能状态，从而找到系统存在的问题。

很多 DBA 试图通过各种价格昂贵的工具来定位数据库性能问题，其实他身边就拥有 STATSPACK 这个功能强大并且免费的系统性能工具。我曾经阅读过大量 DBA 发来的 STATSPACK 报告，并且在解读报告后和他们探讨数据库存在的性能问题。总是有人问我，你是怎么看出来的？这是一个十分复杂的问题，解读 STATSPACK 报告需要对 STATSPACK 报告的各种系统性能指标有比较深入的了解，也需要丰富的数据库管理经验，最重要的是需要有很大的耐心，能够把厚厚的 STATSPACK 报告像看王朔的小说一样津津有味地读完，需要有十分专业的知识和认真的态度。

对于 DBA 来说，随着 Oracle 新版本的推出，STATSPACK 报告越来越容易读了，这对 DBA 来说是一个福音。越来越多的指导性数据出现在报告里，使 DBA 不再像对早期的 STATSPACK 报告或者 BSTAT/ESTAT 报告一样一筹莫展。解读 Oracle 9i 的 STATSPACK 报告是十分有趣的，特别是对于 Oracle 9i 新增的几个缓冲区设置指导图表。Oracle 10g 提供了更简单易用的 AWR。AWR 是和 STATSPACK 类似的工具，基本上可以看做是 STATSPACK 的升级版本。虽然 Oracle 10g 中依然保留了 STATSPACK 这个工具，不过很少人舍弃 AWR 继续使用 STATSPACK。“Oracle 10g 是一个易于管理的数据库”，Oracle 公司经常这样为这个全新的数据库宣传。单从 AWR、ASH 和 ADDM 来看，我觉得这个宣传是完全可信的。

目前有很多对 STATSPACK 信息进行分析的工具软件，有免费的也有收费的（当然价格十分昂贵）。作为专业人士使用的信息来说，这些工具能够达到的数据分析能力十分有限，远远没有 STATSPACK 报告本身所传达的信息丰富。Oracle 公司有一个解读和分析 STATSPACK 报告的工具，只要输入 STATSPACK 报告，就能够输出一份该报告的解读结果。这是我目前见到的最好的 STATSPACK 报告分析工具，但是这份报告传递出的信息还是没法和原始报告相比的。

在今后几个章节的“优化小技巧”中，我将会和大家一起讨论 STATSPACK 报告，目的是通过实例来阐述如何解读 STATSPACK 报告。我将通过解读 STATSPACK 报告，系统地介绍 Oracle 数据库性能优化。因为我认为，Oracle 数据库的优化工作，本来就是应该围绕着 STATSPACK 报告进行的。

对于 Oracle 10g 或者更新的数据库，除了 STATSPACK 以外，DBA 还可以使用 AWR 和 ASH 报告来对数据库进行分析。AWR 报告的内容和 STATSPACK 报告类似。

优化小技巧 STATSPACK 报告头的解读

STATSPACK 报告的第一部分是报告头，其中包含数据库实例的信息以及报告采集的时间等信息。

```
STATSPACK report for

DB Name         DB Id    Instance     Inst Num Release     Cluster Host
------------ ----------- ------------ -------- ----------- ------- ------------
TEST         3209088040 test                 1 9.2.0.4.0   NO      hpux-01

            Snap Id     Snap Time      Sessions Curs/Sess Comment
            ------- ------------------ -------- --------- -------------------
Begin Snap:                          2 12-Jul-04 12:03:05      319     184.1

  End Snap:                         11 12-Jul-04 14:07:49      314     188.5

   Elapsed:                            124.73 (mins)

Cache Sizes (end)
~~~~~~~~~~~~~~~~~
               Buffer Cache:     4,000M      Std Block Size:8K
           Shared Pool Size:       304M          Log Buffer:1,024K
```

- DB Name：报告的数据库的名称。
- DB Id：报告的数据库的 ID，是一个唯一性的编码。
- Instance：数据库实例的名字。
- Inst Num：实例的编号。如果不是 RAC 系统，一般是 1。
- Release：数据库的版本。
- Cluster：是否是 OPS/RAC 环境。
- Host：报告的数据库主机的名称。

- Snap Id：STATSPACK 状态快照（Snap）的编码。
- Snap Time：STATSPACK 状态快照的时间。
- Sessions：状态快照当时的数据库会话数量。
- Curs/Sess：每个会话打开的游标数量。
- Cache Sizes：在终止快照时各种缓冲区的状态。
- Buffer Cache：各种数据块缓冲区的大小，上面的例子是 4 000MB。
- Std Block Size/Block Size：默认块大小，也就是 8i 的块大小（block size）。

6月12日 阴　谋

今天早上起来发现系统运行很正常，CPU 较为紧张的收费系统的 R 队列一般都在 20 以内，CPU 使用率基本上在 80%多，WIO 也在 10%～20%之间。从平均事务响应时间来看，今天上午的平均事务响应时间比优化实施前提高了大约 40%。

我走到小孙的电脑边，问小孙最近 BBS 上反馈怎么样。小孙打开 BBS，发现这几天 BBS 上冷清了不少，甚至有些人在夸系统最近有所改善，希望我们再接再厉。

10 点多有一个营业厅打电话投诉说系统速度很慢，没法工作了。当时把大家都吓了一跳，老于急忙跑到 OMS 服务器上去看实时监控的信息，这几天我们一上班就把几个主要的 OEM 图表打开了，并且开启了录像。一旦有人投诉性能问题，我们立即就可以通过回放来观察系统，找到问题的原因。老于看了半天，没发现有任何异常，系统的各项指标都很正常。

再次打电话回营业厅确认，那边的反馈是现在系统的速度有所改善，但是那个营业员还是感觉系统很慢。我让小孙问问他们营业厅其他的终端是不是都很慢，那边的回答是只有她这个终端慢，其他终端都很正常。听到这个消息，大家都松了一口气，马上把这个服务请求转到做桌面服务的公司去了，估计又是病毒之类的东西搞的。

虚惊一场后，我们几个一起坐在吸烟区喝茶。今天我嗓子不太好，所以没有接老于递过来的烟，只有老于一个人在那里吞云吐雾。第一次调整结束了，有经验也有教训，不过我们没有时间来总结这些。开发商的态度一直是我们担心的问题，这可能成为本项目成败的关键。我们实施优化前曾经考虑过的那个问题真的成了现实，开发商下一步可能会放慢 SQL 优化的进度。如何能给开发商更大的压力，使他们能够在 6 月 20 日之前完成所有的应用优化工作，使项目能够按时验收呢？大家想了半天也没有提出什么好的建议，三个郁闷的人只好先去吃了午饭。

下午甲方的张工他们过来了，主要是和我们讨论下一步优化的方案。张工今天比较兴奋，从昨天和今天上午系统的情况看，系统性能改善十分明显，BBS 上甚至出现了很多正面的消息，很多人都在感谢 IT 部门做了件好事，希望系统能够持续好下去。上午张工甚至在 BBS 上回了几个帖子，对大家的支持表示感谢，并且表示优化还会继续下去，系统会越来越好。

老熊下午去了开发商那边，给他们进一步讲解那个 SQL 去掉占位符后的处理方法。我和老于陪着张工他们两个在楼梯口的吸烟区聊了好久。我提出目前最担心的是系统好转会让集成商觉得目的达到了，因此放松了 SQL 修改。如果开发商不完成我们提交的几十个 SQL 的修改，那么这个优化项目功亏一篑，虽然目前系统性能可以有所提升，不过这种好转无法持续较长的时间。

目前我们的优化虽然有点成果，不过随着系统数据量的增长以及新业务模块的上线，我们获得的优化成果很快就会被抵消，出不了半年，问题就会再现。本来这个问题是应该在昨天的例会上谈的，但是我怕这个问题提出来后大家都很尴尬，所以就没有说。

张工一直对开发商的态度感到不满，听说后感觉问题十分严重，立即和孙主任联系，把我们的担心告诉了孙主任。孙主任让我接电话，问我为什么昨天会上不说这个问题。我说会上讨论这个问题不会有什么结果，最后可能会形成扯皮。现在已经 6 月 12 日了，我们预期是 6 月 15 日集成商完成 SQL 的修改，并且于 6 月 20 日前上线，最迟也不能晚于 6 月 25 日上线，因为 25 日后将是账期处理，再上线新补丁风险很大。而从目前来看，集成商还是不肯对这两个时间点进行确认。

孙主任问我这个问题如何解决，我说如果现在性能不那么好，就可以压开发商了，如果前几天我们不做优化就好了。孙主任说可惜优化已经做了，总不能退回去吧。我说："如果由于某个新开通的模块导致系统负载增加，系统性能又变坏了，那么就要轮到开发商着急了。"孙主任心领神会，说："小白，你小子够可以的，这种招都想得出。那怎么让现在系统的性能变差？"我说："前阵子不是有些模块由于性能问题上线后又被迫下线了吗？如果现在这些模块必须马上上线，那么系统压力估计又很快要冲到 100%了。"

孙主任会心地笑笑说："小白，这方面你是我老师，我真要好好和你学学。"

我说："老人家说过，与人斗其乐无穷。我这也是被逼出来的，做优化的，这种问题是经常碰到的，不无耻一点就没法生存了。"

最后大家决定由孙主任负责发个通知，就说由于优化取得了一定的效果，上个月停用的几个模块可以恢复运行，各营业厅在 6 月 13 日可以恢复使用这几个模块。因为有 1 个月没有开放这几个模块了，而且月底又是半年绩效考核的时间，一旦这几个统计模块一开放，估计会有大量的营业员使用这个模块来对自己的业绩进行统计分析。这可能对系统形成较大的压力，甚至使系统的性能比优化前还糟糕。不过随之而来，孙主任肯定会承受一定的压力。如果能顶住这几天的压力，开发商肯定会坐不住的，业务部门甚至会把问题捅到集团公司去。

放下电话，大家都感到有点好笑，采用这样的方法来压开发商修改程序，真是第一次碰到。正在这时，孙主任的电话又打进来了，他问我："你是否确认以集成商的能力，这些 SQL 是能够在 6 月 15 日之前修改完毕的，会不会修改的工作量太大了？"我告诉他，我们都已经把优化方案写得很清楚了，如果真心要修改，顶多三五天时间，就可以全部改完测试完。孙主任说，如果这样，我就去操作了，大家一定要给集成商足够的压力。

看样子下面头痛的应该是集成商了。明天的 BBS 上应该又是骂声一片了。今天和孙主任的交流十分到位，通完电话我自己都觉得自己很了不起。正沉浸在兴奋之中，我的手机响了，是一个很陌生的电话，沈阳本地的。

原来电话是 HP 维修站打来的，说我的电脑换了主板还是无法启动，看样子是 CPU 出问题了，CPU 目前沈阳没有货，必须从北京调货，可能要一周左右的时间才能修好。再有一周时间，估计我都已经回深圳了，所以我决定不修了，干脆带回深圳再说吧。

打车去三好街取回电脑，我把以前拆下来的硬盘重新装好，顺便按了一下电源开关，电源灯居然亮了，随后电脑就启动起来了。我真是服了维修站的这帮人了，又是换主板又是换 CPU 的，原来电脑根本就没有坏。

今日点评

本书写到这里，基本上也算是到高潮了。今天的主题是阴谋，实际上可以算是阳谋，因为其目的是好的。干脆今天给大家分享一下我做性能优化的一些粗浅的体会吧。

算起来，从第一次给客户优化系统到现在也有 10 多年了。在这些年里，接触过不同的客户、不同的系统，对于优化的体会也越来越深刻。刚开始做优化时，总是希望找出系统中所有存在问题的地方，然后一个一个地进行调整。由于对 Oracle 基本原理的认识不够，并且对优化的认识也仅限于调整不合理的部分这种浅层次上，所以往往做出一些事与愿违的事情来。对于系统优化，今天总结一下最有感触的几点和大家共享。

(1) 优化是基于目标的，我们的最终目的是达到一个目标，而不是优化本身。目标的合理性决定了优化项目的成败。刚刚开始给用户优化时，我会把所有能够调整的东西一次性全部调整完毕。哪怕有些调整给系统性能带来的好处不到 0.1%。对于生产系统，不确定因素十分多，而很多参数方面的调整本身就是双刃剑，如果你无法预期其调整的影响，那么这种调整是存在风险的，在实施时就应该慎重考虑。现在我做优化的项目，往往会根据用户的优化目标，然后在此基础上进行分析，制订方案，实施的结果虽然一般会超出客户的期望，但是我不会在生产系统上做一些没把握的事情。锦上添花的事情有时候也要考虑是否值得，因为弄不好，锦上添花会变成画蛇添足。

(2) 1+1 不一定大于 1。在优化过程中，抓住主要矛盾，解决主要问题，而不要胡子眉毛一把抓。很多调整之间有关联性，甚至是互斥的，不合理的调整可能带来更坏的结果。

(3) 客户需要的是系统的优化，而不仅仅是数据库的优化。在客户的目标里，看到的是一个系统，而不是一个孤立的数据库。在 10 年前，我可能会说，操作系统的问题你们还是找一下厂家。而现在，我会对客户说，你放心，我们做的是系统优化。

(4) 用适当的方式和应用开发厂商配合。很多优化项目由于无法和应用厂商有效的配合，其效果大打折扣。因为应用是和系统性能关系最为紧密的。如果应用开发厂商不能很好地配合，那么优化项目将举步维艰。如果你和开发厂商说“你这个 SQL 开销太大，需要修改一下”，那么得到的可能就是强烈的反对。如果你说“这个 SQL 开销太大，我给你们提供了几个方案，第一是……，第二是……”，这样你很可能会得到比较好的结果。你是优化专家，找出几个 Top SQL 这样的工作不需要专家来完成，而专家的职责不仅仅是发现 Top SQL，而是如何解决掉 Top SQL。

(5) 不要相信什么优化规则，实际上并没有条条框框限制你，实现目标的任何方法你都可以使用。对于一个初级 DBA 来说，老 DBA 可能会告诉你什么是对的，什么是不对的。而对错都是相对的，如果你已经深刻地理解了 Oracle 以及系统优化，那么就像金庸小说里的内功高手一样，

无招胜有招。

优化小技巧 分析 STATSPACK 的负载数据

Load Profile 对于我们了解系统的负载情况帮助很大。经常有人问我："STATSPACK 报告中的某个指标是×××，是不是太高了？"实际上，某个指标是高是低必须依赖于系统的负载情况，因此了解系统的总体负载情况十分关键。下面是一个例子：

```
Load Profile
~~~~~~~~~~~~                            Per Second       Per Transaction
                                   ---------------       ---------------
                  Redo size:            102,614.22              5,270.76
              Logical reads:             86,915.82              4,464.41
              Block changes:                621.01                 31.90
             Physical reads:              1,767.73                 90.80
            Physical writes:                141.67                  7.28
                 User calls:              1,979.58                101.68
                     Parses:                293.79                 15.09
               Hard parses:                   7.43                  0.38
                      Sorts:                 23.28                  1.20
                     Logons:                  0.20                  0.01
                   Executes:                461.41                 23.70
               Transactions:                 19.47

  % Blocks changed per Read:    0.71    Recursive Call %:             26.70
 Rollback per transaction %:    1.35       Rows per Sort:           1012.48
```

Load Profile 标志当前系统的负载情况。对于同一套数据库系统，如果应用没有大的改动，那么 Load Profile 的变化基本上可以反映出系统总体负载情况的变化。Load Profile 中的 `Per Second` 是每秒钟的数据，`Per Transaction` 是每个事务的数据。每秒钟的数据反映系统的总体负载，每个事务的数据反映应用相关的数据。

- Redo size：系统产生重做记录的字节数。通过 Redo size，我们大体上可以看出系统的负载情况。一般来说，这个值越大系统的负载也相对越大。
- Logical reads：系统产生的逻辑读的块数。逻辑读的大小可以看出数据库消耗的系统资源，特别是 CPU 资源的情况。逻辑读越大的系统消耗 CPU 也越高。
- Block changes：块变化的数量。块变化不一定是逻辑读，一些事务操作也会产生块变化。
- Physical reads：物理读的数量。物理读的值越高，说明系统对于 I/O 的负载越大。如果某个系统的物理读突然变高，就要查查是不是某个应用走了全表扫描了。
- Physical writes：物理写的数量。
- User calls：用户调用的数量。
- Parses：SQL 分析的数量。
- Hard parses：硬分析的数量。
- Sorts：排序的数量。

- Logons：登录的数量。
- Executes：SQL执行的数量。
- Transactions：系统中每秒钟事务的数量。
- % Blocks changed per Read：每次读取数据块后被修改的百分比。
- Recursive Call %：为了执行SQL，系统额外需要的SQL称为递归调用（Recursive Call）。这类调用的比例越低越好。
- Rollback per transaction %：回退事务的百分比。
- Rows per Sort：每次排序的记录数。本例中每次排序的记录数比较多。

优化小技巧　重要的命中率指标

虽然说命中率指标不能作为性能分析的主要依据，不过其参考价值仍然是十分大的。特别是对于某类系统（比如OLTP系统），命中率指标还是有一定规律的。对于相同的一套系统，在较长的时期内，其命中率指标基本上会稳定在一个基线上，如果超出了基线的范围，那么肯定系统出现了某种不确定的变化。下面通过一个实例来说明各个指标：

```
Instance Efficiency Percentages (Target 100%)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
            Buffer Nowait %:   99.89       Redo NoWait %:            100.00
            Buffer  Hit   %:   98.02    In-memory Sort %:             99.95
            Library Hit   %:   99.23        Soft Parse %:             97.47
         Execute to Parse %:   36.33         Latch Hit %:             99.90
Parse CPU to Parse Elapsd %:   10.38     % Non-Parse CPU:             95.72

 Shared Pool Statistics          Begin     End
                                ------  ------
             Memory Usage %:    80.10   79.53
    % SQL with executions>1:    87.35   90.32
  % Memory for SQL w/exec>1:    99.04   99.27
```

下面列出了一些重要的数据库命中率指标。

- Buffer Nowait：数据库缓冲区访问无等待的比例，目标100%。
- Buffer Hit：数据库缓冲的命中率，目标100%。
- Library Hit：库缓冲的命中率，目标100%。
- Execute to Parse：（1−Parse/Execute）%，目标100%。
- Parse CPU to Parse Elapsd：分析使用的CPU与分析消耗的总时间的比值，目标100%。如果这个值比较低，说明在分析过程中不仅仅消耗了CPU资源，还有很多等待事件，比如等待共享池对象、闩锁等。
- Redo NoWait：Redo写无等待的比例，目标100%。
- In-memory Sort：内存排序的比例，目标100%。
- Soft Parse：软分析的比例，目标100%。
- Latch Hit：闩锁命中率，目标100%。

- % Non-Parse CPU：除去分析以外的 CPU 的使用率，目标 100%。从这个指标可以了解分析消耗了多少 CPU 资源。
- Shared Pool Statistics：共享池使用情况。
- Memory Usage%：共享池使用百分比。如果太低，说明共享池可能不足。不过这个值不是绝对的，对于不同的系统其表现不同。这个值接近 100%并不一定就说明共享池已经不足，需要扩充了。
- % SQL with executions>1：执行次数大于 1 的 SQL。该值越高说明 SQL 的重用率越高，越高越好。
- % Memory for SQL w/exec>1：执行次数大于 1 的 SQL 占用的内存占所有 SQL 的比例。

6月13日 演　　戏

今天沈阳这边只剩下我和老于两个人了，前两天老肖就因为浙江的事情离开了，今天上午老熊也回深圳去了。小齐昨天开完会就急急忙忙地赶到深圳去了。旭日印刷厂那边急着要我们尽快进场，小齐和老熊估计又要在那里折腾半天。如果旭日那边吃紧，老熊这回走了之后也可能不会再回沈阳了。在老熊走之前，我们两个已经把所有的 SQL 优化方案搞好了，并且提交给了老于。项目做到这个阶段，基本上我们该做的事情都已经做完了，剩下的就完全依靠开发商的支持了。不知道昨天和孙主任安排的这一出苦肉计是不是能起到作用。

按照昨天的布置，我和老于今天早上出门晚了一点，到办公室时已经快 10：00 了。一到办公室，我们就发现小孙正在焦头烂额地接电话，估计昨天孙主任的招下得够狠的。小孙是集成商常驻客户这边的技术人员，主要负责解决现场的问题。6 日优化实施后，电话投诉的量大幅度下降，小孙的压力减轻了不少，用他自己的话说是度过了在沈阳最美好的一段时光。今天估计小孙又要重温以前不堪的日子了。

我装作什么都没看见，打开电脑，拿出茶杯装好茶叶，拿着茶杯跑到 OMS 服务器上，打开 OEM 的性能分析工具。随后就到旁边的开水房泡了一大杯水。

回到办公室，放下水杯，我的笔记本电脑还在启动。这破本子是该换了，还是个赛扬的 1.6GB 的本子。虽然装了 2GB 内存，还是够慢的。和往常一样，在电脑启动的过程中，我跑到 OMS 服务器上看了一眼。一望过去，是一片惨不忍睹的指标，平均事务响应时间高达 3 秒多，比昨天慢了足足一倍多。从 OEM 的健康性检查图表上看，系统存在大量的 db file sequential read 和 db file scattered read 等待，CPU 的曲线基本上是一条贴近 100%的直线。

这几个统计查询模块是营业员查看和统计自己的业绩的，作为绩效工资发放依据的绩效考核指标都必须通过这几个模块来查询。由于绩效指标十分多，所以需要做很多大型的查询才能算出一个营业员在某个时间段内的绩效。对于这几个模块，当时我提出过优化建议，一是对营业员的绩效统计，能否先形成日统计数据，这样跨时段统计数据就可以在日统计基础上二次汇总获得。这个方案由于程序改动过大，被开发商否决了。后来老熊也提出过，如果不能进行日统计汇总，那么是否能够限制一下统计的时间范围。因为现在在统计时间区间上没有任何限制，很多营业员会选择半年甚至一年的时间范围来统计，甚至有些人选择几年的数据进行统计。这种情况下，很多查询都会对很大的生产日志表进行全表扫描，预先设计好的索引根本无法发挥作用。如果能够限定查询时间跨度不能超过 1 个月，那么绝大多数查询都可以使用索引了。营业员完全可以通过

多次查询获得自己的绩效数据，然后自己手工去进行季度、半年和年度汇总。这种优化措施我们以前在好多系统中采用过，效果还不错。不过老熊提出的这个优化建议还是被开发商拒绝了，范总说如果是这样，营业员可能会有意见。

由于系统资源紧张，这几个使用率极高的模块已经停用一个多月了，营业员发了绩效工资都无法到系统中去确认。按照我们和孙主任约定好的，这几个模块昨天晚上打开了，并且今天一早孙主任以信息中心名义发了一个模块正式重新启用的通知，一大早就通过电子邮件、OA 系统、BBS、系统启动提醒等多种形式传达出去了。收到通知的全省上千名营业员肯定会大量查询，系统不出问题才怪。

看到这一切，我吃惊地大叫："怎么今天系统又这么差了，出什么事了？"小孙正在接电话，他马上把话筒捂住，说："是的，今天一早就有很多电话，都是投诉性能问题的，我正在处理呢。白哥，于哥，你们快帮着看看。"

老于也装模作样地在 OMS 服务器上翻着，一边看一边还嘀嘀咕咕的："怪了，怎么今天业务量这么大，好多没见过的 SQL，看样子 CPU 是撑不住了。老白，马上向孙主任汇报吧。"

我和小孙说，看样子必须向孙主任汇报了，小孙，你也向范总汇报一下吧，你们的应用要抓紧改了，否则撑不过这个月底了。说完，我就接通了孙主任的电话："孙主任，今天系统负载太大了，您那边是不是有什么新业务上线啊。哦，原来是这样啊，这个模块能不能不开，哦，不行啊，如果这个模块必须开的话，那么开发商那边的 SQL 修改要提前了，否则这几天会撑不住的。好的，我把电话拿给小孙，他正在向范总汇报这个事情。"

小孙还在结结巴巴地向领导汇报，我说："小孙，孙主任找你，我先和范总说两句吧。"说着我把电话递给小孙，顺手把小孙的电话接了过来。"范总，您好。看样子必须您老亲自出马摆平了，SQL 估计要提前上线了，否则系统撑不住了。什么，那几个模块能不能再停一下，不行了，已经停了快 2 个月了，营业员必须知道自己的考核指标，否则发绩效工资时大家又会闹了。营业厅那边闹得很厉害，孙主任也压不住了。范总，其实那几个 SQL，你找两个人，估计两三天也搞定了。您这边再按兵不动，孙主任和我这边实在是挺不住了。好的，具体你还是和孙主任通个电话吧。另外，老兄，实际上那几个绩效模块，你只要限制一下查询时间的跨度，就不会出现现在的情况了。做这么点调整，小王他们那边随便出个人，半小时之内就搞定了。"看到范总有点急了，我觉得这回可能有戏。

那边小孙也已经和孙主任通完电话了，也是一脸的凝重。我安慰小孙，"和公司好好协调一下，别自己一个人扛着。真出了问题，公司肯定不会表扬你为公司坚持原则，出了问题，责任肯定是你一个人担着，你有必要替公司去承担这么大责任吗？还是把这里的真实情况向范总好好汇报一下，让他下决心加快 SQL 优化，如果这回优化成功，那么你今后也轻松多了。"旁边的投诉热线又响了起来，小孙又忙着接电话去了。看样子，今天是消停不了了。希望能通过这件事给范总这个老狐狸足够的压力，甚至一劳永逸地解决问题。

过了没多久，小王打电话过来通知我们，说中午有一个紧急补丁要发布，估计需要停机 1 个小时。我问小王，是什么补丁这么急着发布。小王说："白哥，还不是你给我找的事情，范总刚

才打电话来，让我马上把绩效考核查询的时间区间做个限制，不能跨月度进行统计查询。”范总不愧是在江湖上混了那么多年的高手，都到这个时候了，他还能这么冷静地拆招。

和老于商量了一下，我马上把这个情况和孙主任通报了一下，范总还是想修修补补，不想按照我们的思路去做。

孙主任想了想说如果中午发布了修改后的程序，那么今天下午系统是不是能够得到缓解。我说如果修改了程序，系统应该能撑得住，系统情况会比上午好很多。

“那怎么办，小白？”孙主任也对范总没有办法了。

“孙主任，只要你那边能顶得住，我肯定有办法。不过今天下午可能还会有大量的投诉，你们部门的压力不小，具体怎么做你就不需要知道了，出了问题你帮我兜着就行了。”看到范总这个样子，我也只能下狠招了，我找出几个以前性能很差的统计模块，改了改，让这几个模块每隔两三分钟跑一次，这样就可以消耗大量的系统资源，下午系统性能想好也不那么容易了。

中午小王发布补丁前，我就把这几个模块的定期执行脚本放到了 OSW 的脚本里，每次 OSW 采集操作系统数据时，就会顺带跑一下这几个大查询。果然和我想象的一样，补丁发布后，系统性能并没有出现任何好转。小王让我分析了一下系统，我说：“小王你发布的补丁是起作用了，SQL 的执行效率提高了几十倍，不过架不住查询太多了，整个系统的资源瓶颈太严重，所以你的这个修改也只是杯水车薪。”

下午 3 点多，我觉得小王应该已经向范总汇报过情况了，就打电话告诉孙主任，需要他出马再去将范总一军了。果然，我和孙主任的电话打完还不到半个小时，小王就打电话过来说：“白哥，范总已经通知我安排资源加快 SQL 优化了，不过我们最近人手实在是特别紧张，我只能安排两三个人出来，你明天能不能到我们这里来上班，教教他们该怎么做。我们范总明天会和你联系。”

我长出了一口气，范总这个老狐狸终于开始和我们配合了。我马上打电话告诉孙主任，平安无事了。

今日点评

项目做到这种地步，需要通过这种手段来逼开发商改应用，也确实是够难受的。不过这并不是老白的虚构，而是真实经历。不过实际的故事不是发生在这个项目里，老白把这段写进来的目的是为了制造一个小的悬念。实际上，本书虽然以沈阳这个项目为蓝本，但讲述的仍然是一个虚构的项目，因此参与过沈阳这个项目的人请原谅我的艺术加工。

优化小技巧　Top 5 事件

在做性能优化的过程中，抓住主要矛盾是十分重要的。数据库性能方面的指标十分复杂，优化过程中也是千头万绪。因此解决主要的性能问题，尽可能少地改动系统就能达到优化目的是十分重要的。STATSPACK 报告中的 Top 5 事件就是我们抓住系统关键问题的一把钥匙，Top 5 等待

事件是系统中最重要的等待事件。解决 Top 5 等待事件是进行数据库优化的关键。从 Oracle 9i 开始，Top 5 等待事件被改为 Top 5 Timed Events。因为本节可能包含 CPU time 这类非等待事件，所以 Top 5 Timed Events 更为贴切。下面是一个例子：

```
Top 5 Timed Events
~~~~~~~~~~~~~~~~~~                                                    % Total
Event                                            Waits    Time (s) Ela Time
-------------------------------------------- ------------ ----------- --------
db file sequential read                        11,329,122     153,771    79.27
CPU time                                                       22,660    11.68
buffer busy waits                                 744,501      11,596     5.98
db file scattered read                            173,282       4,602     2.37
db file parallel write                             30,423         392      .20
          -------------------------------------------------------------
```

在一个性能比较良好的系统中，大多数数据库的 db file sequential read 和 db file scattered read 可能是排在最前面的等待事件。从这个报告上看，db file sequential read 占了整个等待事件的 79.27%，因此后面需要详细分析一下这个等待事件，这个事件是由于单块读引起的。排在第二位的是 CPU time。对于一些等待事件比较小或者高开销 SQL 比较严重的系统，这个等待事件可能会排在第一位。这个事件是由于 SQL 执行、分析等消耗 CPU 资源的行为带来的。buffer busy waits 是由于热块等待造成的等待事件。在这里，这个等待事件为 6%左右，说明系统中还是存在一定的热块冲突。db file scattered read 等待事件是由于全表扫描或者全索引扫描引起的，其特点是多块读，对某个对象进行扫描。不能片面地认为 db file scattered read 就是全表扫描，存在这个等待事件并不说明 SQL 一定存在问题，因为有些应用进行全表扫描或者全索引扫描是正常的。db file parallel write 是写入数据文件产生的等待。在这个系统中，这个等待事件所占比例很低，可以不考虑。

本书将会给出一些常见的 Oracle 事件以及状态值的定义，不过更多的信息需要 DBA 在日常工作中积累。*Oracle Wait Interface* 是一本很不错的关于 OWI 的书，建议大家阅读。

优化小技巧　常见事件清单

常见事件清单如下表所示。

名　称	说　明
CPU used by this session	统计调用发起开始到结束的CPU时间片的数量。每个计数代表一个CPU周期，也就是10ms。不过如果有一个调用不足一个CPU周期就执行完了，那么统计值时起始时间和结束时间是相同的，这样会计算为0这个计数基本上可以代表Oracle数据库消耗的CPU资源，不过计算时要注意单位是厘秒（cs），乘以10就可以换算成毫秒（ms）。比如平均每秒这个值是782.1，表示Oracle消耗了7 821毫秒CPU时间，如果这个系统是一个16颗CPU的系统，那么这个值可以说明Oracle消耗了超过50%的CPU资源。不过由于部分小的调用可能由于消耗时间小于10ms而没有计算进去，实际的使用率可能略高于通过这个值计算出来的。一般来说，大多数调用消耗的CPU都会大于10ms，所以这个值还是能够基本反映出Oracle对CPU资源的开销

（续）

名 称	说 明
CR blocks created	当前块被克隆后用于创建CR（Consistent Read）块。被克隆的主要原因是缓冲区被非兼容的模式占用，如果单位时间内CR blocks created值比较高，说明数据库中对某些数据块的修改和访问比较频繁。如果这些访问集中在某些热块上，那么可能会形成较为严重的buffer busy waits，在RAC环境中，可能还会导致全局热块冲突。如果这个值比较高，那么应该关注buffer busy waits以及cache buffer chains闩锁等
current blocks converted for CR	一个当前的buffer在使用前生成了CR
DBWR buffers scanned	当某些触发条件发生时，DBWR会在LRU链的冷端开始扫描脏块，组成DBWR batch，这个值统计的是DBWR在LRU上扫描的buffer的总数，包含脏块和干净的块。这个值除以DBWR LRU scans就是每次扫描查找的数据块的数量
DBWR checkpoint buffers written	检查点时DBWR写入的脏块数量。如果在单位时间里这个值比较高，说明系统中数据块的变更较为频繁
DBWR free buffers found	DBWR从LRU链中扫描buffer时发现的空闲的buffer的数量，除以DBWR make free requests就是平均每次DBWR在收到DBWR make free消息时扫描LRU链找到的空闲buffer的平均数，这个平均数一般会比较少
DBWR make free requests	DBWR收到的make free消息的数量。如果某个前台进程无法找到空闲的buffer时，会向DBWR发出make free消息。如果单位时间内这个值较高，说明DB Cache可能存在不足的现象
DBWR summed scan depth	DBWR扫描LRU链查找脏块时，查找的buffer的数量。这个数越大，说明LRU链尾部的脏块数量越少。从Oracle 8i开始，由于LRU链的算法发生了变化，所以如果LRU链尾部的热块比较多，也可能造成这个值较大
DBWR timeout	DBWR idle超过一个特定值，该值就会加1。如果该值较高，说明Buffer Cache中的数据变化较小，需要写入磁盘的脏块数量极少
DBWR transaction table writes	DBWR写入的回滚段头的数量。该值比较高说明有较多热块正在被写入，而大量用户进程在等待这些块写入完成
DBWR undo block writes	DBWR写入回滚段的数据块数量
DDL statements parallelized	DDL操作并发执行的计数
DML statements parallelized	DML操作并发执行的计数
background checkpoints completed	后台进程完成的checkpoint的数量
background checkpoints started	后台进程启动的checkpoint的数量，可能比上一个状态的值大一些。这个状态只包含重做的checkpoint，不包括其他类型的checkpoint，比如`OFFLINE`文件或者`BEGIN BACKUP`或者`ALTER SYSTEM CHECKPOINT LOCAL`命令等
branch node splits	由于插入数据导致的索引枝节点分裂的数量。这个值较高说明目前存在索引产生了较多的枝节点分裂，可能某张表上的某个索引字段变化十分频繁，这种频繁的变化可能对某个应用的性能有较大的影响
buffer deadlock	DB Cache死锁的数量计数。如果单位时间内该值较高，可能DB Cache存在性能问题，或者存在某些bug
buffer is not pinned count	当访问一个buffer时，这个buffer已经释放的数量。只用于Oracle内部调试，并不说明性能问题
buffer is pinned count	当访问一个buffer时，该buffer已经被PIN住了。如果单位时间内这个值比较高，说明可能存在热块
change write time	当前块的变化被写入redo的时间，单位为厘秒（cs，10ms）。该值一般不会太大。如果太大，就需要分析

（续）

名　　称	说　　明
commit cleanout failures: block lost	在COMMIT时，准备做块清理操作时，发现不能找到正确块的次数计数
commit cleanout failures: buffer being written	Oracle在COMMIT时，去清除buffer时，发现这个buffer正在被其他会话写入的计数。如果该值比较高，可能说明存在热块
commit cleanout failures: callback failure	Oracle在COMMIT时，做清除操作时调用回调函数返回FALSE的计数
commit cleanout failures: cannot pin	Oracle在COMMIT时，做清除操作时无法PIN到这个buffer的计数。有可能在准备清理时该buffer又被其他会话PIN住了。如果该值较高，可以查看DB Cache相关的情况，包括DB Cache的大小、命中率、相关闩锁的命中率以及热块争用的情况
commit cleanout failures: hot backup in progress	Oracle在COMMIT时，做清除操作时发现正在做热备份的计数。此时这个buffer在被修改前，必须先被写入Log Buffer，以确保数据库恢复时不会产生块断裂
commit cleanout failures: write disabled	Oracle在COMMIT时，做清除操作时发现数据库的写操作暂时被关闭了。这种情况出现得很少
commit cleanouts	在COMMIT时做块清除工作的计数，无论成功与否计数器都会加1
commit cleanouts successfully completed	COMMIT时成功完成cleanouts的计数。这个值和上一个值相比，两个值应该较为接近（这个值略低一些）。如果这两个值相差太大，需要分析DB Cache是否存在过小的问题，或者应用中是否经常对大表进行大数据量的修改操作
consistent changes	数据块提交了undo信息成为CR块的计数。这个值说明了系统中CR块产生的数量。这个值越大，越要注意cache buffer chains等闩锁的情况以及热块对系统性能的影响
consistent gets	一致性读的计数。会话发出的对某个数据块进行一致性读的请求。不能将其与consistent changes混淆。一个CR块产生后，可能被多个consistent gets事件调使使用，因此这个值要比前一个值大得多
data blocks consistent reads - undo records applied	从undo中读取数据，形成CR READ。本计数器记录从undo中获取undo记录的计数。如果这个值较大，说明对于某些修改较为频繁的表的查询和其他操作也很频繁，有可能存在热点表和索引
deferred (CURRENT) block cleanout applications	做延迟块清除操作的计数。在提交时该数据块由于某些原因，某些数据块无法马上做块清除工作。这种情况下，这个数据块就会做延迟块清除，延迟块清除操作可能在下次该数据块被查询时进行。这种情况也导致有时候我们会看到做SELECT操作时也会产生大量的redo
dirty buffers inspected	当某个会话在LRU链的冷端开始查找空闲的数据块时查到一个脏块，这个值就会增加。如果单位时间内该值较大，说明LRU链的冷端存在较多的脏块。出现这种情况有几种可能： (1) 系统中的脏块数量十分巨大，而且DBWR的写入速度不足，从而导致DBWR无法尽快将这些脏块写入硬盘； (2) 部分buffer特别热，并且被更改的频率特别高，从而造成LRU链的尾端存在大量这样的块； (3) 本系统是一个以DML为主的系统，数据块的变更十分频繁。 碰到这种情况，可以关注一下DBWR的性能，并且关注一下DB Cache的命中率及cache buffer chains等闩锁的情况
enqueue waits	等待各种锁的计数

（续）

名　　称	说　　明
exchange deadlocks	当进行两个buffer交换时，发生内部死锁的计数。索引扫描是导致这种交换的唯一因素。如果该值较高，可以检查是否存在十分热的索引（可以通过buffer busy waits分析来定位）
free buffer inspected	从LRU队列的尾部扫描可重用的buffer时跳过的buffer的数量
global cache freelist waits	当ping一个buffer时，由于所有的lock element都被使用而引起的等待
global lock convert time	同步全局锁的转换时间（单位是10ms）。这个值较高说明全局锁冲突较为严重，需要检查cluster interconnect的性能
hot buffers moved to head of LRU	当一个热块到达LRU队列的尾部时，Oracle自动会把这个块移动到LRU队列的头上，以便于使之能够继续被使用。每发生一次这样的操作，这个计数就加一。值得注意的是，从Oracle 8i开始，LRU的算法发生了变化，通过引入TCH计数来确定热块，而不是通过将热块在LRU链上移动来保证热块不被过早地换出。如果热块存在于LRU链的尾部，扫描时发现了热块，会主动跳过，从而保证热块不被过早地重用
immediate (CURRENT) block cleanout applications	获取BUFFER后，立即进行记录清除操作的计数
leaf node splits	当INSERT发生后，导致索引叶节点分裂的次数。一般来说，对于插入较为频繁的系统，这个值一般会比较高。除非出现叶节点热块较为严重的现象，一般来说不需要特别关注该值
logons current	当前登录数据库的计数
opened cursors current	当前打开的游标的数量
opens of replaced files	文件由于不在打开文件缓冲中而重新打开的计数。每个Oracle会话都有一个文件打开缓冲区，保持部分打开的数据文件的句柄，以避免重复打开文件
parse count (hard)	硬分析的统计值。该值需要和parse count (total)对比来看硬分析的比例
parse count (soft)	软分析的统计值
parse count (total)	发生的parse call的总数
parse time cpu	parse消耗的CPU的统计，单位是10ms
parse time elapsed	pase的持续时间，单位是10ms。这个值减去parse time cpu就是parse中等待的时间。如果parse time cpu占整个parse time elapsed的比例较低，说明parse中等待时间过长，可能共享池存在性能问题，需要进行分析
physical reads direct	直接物理读的数量。读时不经过buffer。一般发生这种操作的情况有：排序操作、并行查询操作的从属进程或者预读
physical writes direct	直接写的数量，不经过buffer，直接写入。一般发生这种操作的情况有： ❑ 直接装载操作，比如CREATE TABLE AS SELECT； ❑ 并行DML操作； ❑ 排序操作中的临时表空间写入； ❑ 写入没有缓冲的LOB字段
physical writes non checkpoint	非checkpoint引起的物理写。物理写的发生情况包括checkpoint或者无足够的空闲buffer可用，或者DBWR超时等。一般情况下，这个值会超过physical writes的一半以上，除非是checkpoint十分频繁的系统。如果该值占physical writes的比重比较少，应该进行分析
pinned buffers inspected	当一个用户进程扫描replacement列表，寻找一个可重用的buffer时，发现一个冷块被PIN了或者有一个PIN请求的等待事件。这种情况很少发生，因为冷块很少会被PIN。如果平均每秒该值较大，需要进行分析

（续）

名　　称	说　　明
queries parallelized	并行查询执行的次数统计
recursive cpu usage	非用户调用的CPU时间
recovery array read time	恢复时产生的I/O消耗的时间
recovery array reads	恢复时产生的I/O的次数
recovery blocks read	恢复时读取的数据块的数量
redo entries	当一个redo信息被复制到Log Buffer时，这个计数会增加
redo entries linearized	小于等于于REDO_ENTRY_PREBUILD_THRESHOLD的redo entries的数量。这些redo entries可以并发生成，不需要受闩锁的限制，但是会增加CPU的消耗。在多CPU的系统中，这个值会比较高
redo buffer allocation retries	当申请redo buffer时重试的次数。一般来说，重试发生的原因是redo writer还没完成或者日志切换正在进行。如果该值较高，说明重做日志产生的频率很高，LGWR无法及时刷新Log Buffer，可以考虑加大Log Buffer的大小。Log Buffer一般可以为几兆到几十兆。不过由于很多数据库版本中存在bug，因此不建议将Log Buffer设置得过大。一般来说，30MB～40MB对于绝大多数系统来说都是足够的。如果是由于等待日志切换，那么可能存在的问题包括： ❑ 重做日志文件过小； ❑ 重做日志I/O性能不佳； ❑ 数据文件的I/O性能不佳导致DBWR写入较慢； ❑ DBWR的数量太少，导致DBWR的写入性能不足
redo log space requests	当前活跃的日志文件满了，Oracle必须等待日志切换完成后才能分配重做日志磁盘空间，此时就会产生这个等待。如果SGA的大小和日志文件的大小不匹配，并且系统中的COMMIT操作十分频繁。当日志切换时，DBWR需要把已经提交的脏块也写入磁盘，在这些脏块写入磁盘完成前，日志切换不能完成。在这种情况下，这个统计值会比较高。 如果这个统计值较高，建议检查以下几个方面： ❑ 重做日志文件的I/O性能是否存在问题，如果log file parallel write的平均耗时较大，说明重做日志的写性能出现了问题，建议将重做日志放到性能更好的磁盘上，或者将重做日志文件独立存放，避免I/O冲突； ❑ Log Buffer的大小是否过小； ❑ 应用软件中的提交可能过于频繁，建议采用批量提交的方式，而不要每条记录都提交
redo log space wait time	redo log space requests的等待时间（单位为厘秒），这个值需要和上一个值一起看
redo size	生成的所有redo的大小，单位是字节
redo synch time	redo synch调用消耗的时间，单位是10毫秒（厘秒）
redo sync writes	一般来说redo信息生成后，会被复制到Log Buffer中，并不需要马上被写入redo log文件，而lgwr会周期性将这些数据写入重做日志文件。不过如果事务提交了，那么这些redo 信息必须马上被写入重做日志文件，这个时候redo sync writes计数器会增加
redo wastage	在把Log Buffer数据写入重做日志文件时，计算的Log Buffer空闲的空间
redo writer latching time	LGWR获得每个copy latch的时间（单位为厘秒）。如果该值存在问题，需要检查重做日志文件的I/O性能以及Log Buffer的大小是否足够。这个值只有在log_simultaneous_copies大于0时才有意义

（续）

名　　称	说　　明
redo writes	LGWR将Log Buffer写入重做日志文件的计数
remote instance undo block writes	如果远程实例需要读取某个undo块，需要这个实例先将这个“脏的”undo块回写，这个计数器就会增加
remote instance undo header writes	和上一个值类似，只是写入的是undo header
remote instance undo requests	由于要做CR而从远程实例中请求undo的数量。如果这个值较大，说明RAC中的某些数据块经常在实例间共享，某个实例修改过的数据也正在被其他实例使用。这种情况下，需要留意CLUSTER INTERCONNECT的性能
rollback changes - undo records applied	当用户需要进行ROLLBACK操作时，提交undo记录的数量。当事务需要回滚时，需要从undo中取出数据，并且提交到已被修改过的数据上，这个计数和系统中ROLLBACK的数量以及每次ROLLBACK的记录的数量有关
rollbacks only - consistent read gets	当用户需要进行CR READ操作时，提交的undo记录的数量，此时并没有buffer clear操作发生。当进行一致性读时，也需要从undo中取出相关的数据，然后生成一个CR块，把这些数据提交到CR块上，这时这个值就会增加
rows fetched via callback	回调函数返回的记录数。该统计量仅仅用于内部调试
session cursor cache count	会话游标缓冲区的计数。只有当session_cached_cursors大于0时才有意义。如果这个值已经很接近甚至达到了参数中session_cached_cursors的值，那么说明这个参数可能需要加大
session cursor cache hits	这个值也只有当session_cached_cursors大于0时才有意义。会话直接从会话游标缓冲区中找到某个SQL的计数。通过这个值，可以看出游标缓冲区产生的作用，可以将这个值和parse count(total)进行比较
sorts (disk)	磁盘排序的数量。如果平均每秒该值较高，需要检查一下PGA_AGGREGATE_TARGET参数是否设置过低，或者*_AREA_SIZE是否设置过小（PGA手工管理模式）。检查该值时，同时应该查看一下temp表空间的相关文件的I/O性能。如果temp表空间文件的I/O性能不足，需要加大PGA的配置来减少硬盘排序
sorts (memory)	内存排序的统计
summed dirty queue length	每次写请求发生时，脏数据块链表的长度总和除以写请求的次数。这个统计值越大，说明系统中需要回写的脏块较多
switch current to new buffer	将当前buffer移动到另外一个新的buffer，原有的buffer成为一个CR块，这种操作的次数
table fetch by rowid	通过ROWID访问表，这种操作一般是通过索引访问，还有一种情况是SQL直接通过ROWID去访问
table fetch continued row	如果访问某一行数据时，需要访问多个块，那么这个计数器就会增加。产生这种情况的一个很主要的原因是行链和行迁移。如果某个表的PCT_FREE设置不合理，可能导致更新时产生行迁移，这样就会增加这个计数。 第二种可能性是某一行计数的长度相对块大小来说过大，这样一行数据被存储在多个块中的机会就较大。这种情况下，应该将这种表存放于更大的块的表空间中。 还有一种可能性就是系统中带LOB字段的访问较多，因为大的LOB字段一般来说采用独立的segment存放，所以访问这种数据也会增加这个统计值
total file opens	被instance打开的文件的数量（包含数据文件、控制文件、重做日志文件等）
transaction rollbacks	被成功回退的事务总数
write clones created in background	如果当前的buffer正在被写入，那么后台进程或者前台进程克隆一个新的buffer，使原来的buffer的写入可以继续进行

（续）

名　　称	说　　明
enqueue conversions	表或者行锁的转换统计
enqueue deadlocks	死锁统计
enqueue releases	锁释放统计
enqueue requests	锁申请统计
enqueue timeouts	锁申请超时统计
enqueue waits	锁等待统计

6月14日

转　　机

今天一早范总就打电话给我，希望我们能够派个人和他们现场的项目经理协调，协助他们尽快把12张表连接的SQL修改完毕。范总是个明白人，他也很清楚这个SQL给系统造成的影响。如果能通过这个SQL优化解决系统的主要问题，那么其他SQL改不改都可以了。

其实范总这个思想转变我昨天就听小王说了。不过为了防止发生意外，昨天的那几个大查询还是继续在跑着，今天上午系统的性能虽然没有昨天那么糟糕，但也好不到哪里去。接了范总的电话后，我觉得不会再出什么岔子了，于是把那几个查询停了，然后就去了小王那里。小王正在和几个开发人员布置优化的事情，见到我来了很高兴，说："白哥，我们必须18日前完成修改，所以你们一定要全力支持我们，干脆你这几天就到我们这里上班算了"。

小王已经把我们要求优化的30多个SQL都安排了实施责任人，我就一个一个地和他们确认优化方案。其实我们的优化方案写得十分详细，如何改写SQL都很明确。因此，只花了不到一个小时，几个开发人员就都已经明确了方案。最复杂的还是那个12张表连接的SQL，小王安排了一个工作了四五年的老手小侯来实施。实际上，老熊把生成SQL的一个存储过程都写好了，只要以这个存储过程为核心调整一下，程序就很容易实现了。但是由于这个SQL涉及的业务十分复杂，所以对测试的要求十分高。我和小侯谈方案时，希望他先花一两天时间把测试用例写好，否则一旦应用上线后出现问题就麻烦了。

谈了一圈，大家的任务基本上也明确了。最后大家确定了一下工作量，觉得除了12张表连接那个SQL外，其他程序的修改和测试都可以在18日前完成，19日凌晨可以上线。而那个12张表连接的SQL，小侯也说最晚20日可以完成，21日凌晨可以上线。按照这个进度，虽然比我们的预期慢了一周，但是如果真的能在25日前30多个SQL全部上线，那么本次优化的效果应该十分明显。

早上的事情让我今天心情大好。中午吃饭时和老于讨论下一步的工作，最近这段时间要花更多的时间在文档上了。系统级调整基本已经完毕，看样子效果还不错。SQL优化如果完成，那么完成这个优化项目的预定目标是肯定没问题了。还有就是，在25日前我们要把所有的索引全部重新重建一遍，这几天晚上下班时晚走一会儿，每天重建一部分就可以了，这样就省得客户陪我们加班了。

我们除了完成优化外，在验收前还需要提交7份文档，即《优化工作界面》、《DBA维护工作手册》、《系统性能基线报告》、《系统性能评估指标手册》、《系统优化方案》、《系统性能优化评

估报告》和《优化工作总结》。《优化工作界面》和《系统性能基线报告》这两份文档在前期已经完成了。剩下的文档还没有开始编写，这段时间工作比较紧张，所以也没顾得上写报告，这几天重点要写报告了。老于分了一下工，我负责《DBA 维护工作手册》和《系统优化方案》的编写。这两个文档都有模板，写起来不难，就是量比较大，工作量不小。

下午突然接到青岛那边万科长的电话，他们的系统遇到了很严重的性能问题，响应速度突然下降了很多倍。万科长他们是通过活跃会话数量突然猛增意识到系统出问题的。他们已经有了一个经验性的指标：当活沃会话的数量在 100～150 时，系统是正常的；如果活跃会话的数量超过 200，系统的性能就有问题了。而今天上午一上班，活跃会话的数量就高达 300。随后，投诉电话就不断打进来。万科长他们已经做了 STATSPACK 报告，并且已经发到了我的邮箱里。

立即打开邮箱收邮件，发现 Load Profile 中的信息如下：

```
Load Profile
~~~~~~~~~~~~                            Per Second       Per Transaction
                                   ---------------       ---------------
                  Redo size:            203,292.09              5,850.14
              Logical reads:            110,127.43              3,169.14
              Block changes:              1,256.72                 36.16
             Physical reads:              5,004.87                144.03
            Physical writes:                126.79                  3.65
                 User calls:              3,380.75                 97.29
                     Parses:                560.50                 16.13
                Hard parses:                 13.01                  0.37
                      Sorts:                 31.81                  0.92
                     Logons:                  0.46                  0.01
                   Executes:                800.51                 23.04
               Transactions:                 34.75

  % Blocks changed per Read:    1.14    Recursive Call %:          27.47
 Rollback per transaction %:    1.61       Rows per Sort:        1896.84

Instance Efficiency Percentages (Target 100%)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
            Buffer Nowait %:   99.93       Redo NoWait %:         100.00
            Buffer  Hit   %:   95.47    In-memory Sort %:          99.99
            Library Hit   %:   99.24        Soft Parse %:          97.68
         Execute to Parse %:   29.98         Latch Hit %:          99.75
Parse CPU to Parse Elapsd %:    2.12     % Non-Parse CPU:          96.75

 Shared Pool Statistics        Begin   End
                               ------  ------
             Memory Usage %:   79.88   78.79
    % SQL with executions>1:   86.48   86.74
  % Memory for SQL w/exec>1:   99.28   99.08

Top 5 Timed Events
~~~~~~~~~~~~~~~~~~                                                  % Total
Event                                              Waits    Time (s) Ela Time
-------------------------------------------- ------------ ----------- --------
latch free                                      1,098,363     606,375    76.58
```

```
db file sequential read                  11,615,630      73,616     9.30
CPU time                                                 55,262     6.98
db file scattered read                    2,377,376      26,913     3.40
library cache pin                             3,865      10,408     1.31
```

latch free 占了 76.58%。检查 Latch 争用的情况，发现：

```
                                           Pct    Avg   Wait                 Pct
                              Get          Get   Slps   Time       NoWait NoWait
Latch                       Requests      Miss  /Miss    (s)     Requests   Miss
------------------------ -------------- ------ ------ ------ ------------ ------
Consistent RBA                  295,240    0.0             0            0
FAL request queue                    10    0.0             0            0
FIB s.o chain latch                 194    0.0             0            0
FOB s.o list latch               99,524    0.0    0.5      1            0
SQL memory manager latch              1    0.0             0        2,431    0.0
SQL memory manager worka        500,795    0.0    0.6     21            0
X$KSFQP                              49    0.0             0            0
active checkpoint queue          25,960    0.0             0            0
alert log latch                      23    0.0             0            0
archive control                     300    0.0             0            0
archive process latch               171    0.0             0            0
cache buffer handles          2,868,069    0.1    0.3    125            0
cache buffers chains      1,528,766,244    0.1    0.4 ######   65,503,162   23.5
cache buffers lru chain       1,667,682    0.6    0.2    782   69,088,014    3.8
cas latch                    52,531,336    0.1    0.3   3824            0
channel handle pool latc          6,653    0.0             0            0
channel operations paren         15,341    0.0             0            0
checkpoint queue latch        5,245,583    0.0    0.2      7      817,925    0.0
child cursor hash table       1,617,769    0.1    0.3     86            0
direct msg latch              1,411,172    0.0    0.0      0            0
dml lock allocation           1,714,958    0.2    0.5    304            0
done queue latch                334,052    0.1    0.1      0      350,600    0.1
dummy allocation                  6,911    0.0             0            0
enqueue hash chains           4,946,060    0.1    0.5    752            0
enqueues                      4,910,923    0.2    0.5    815            0
event group latch                 3,325    0.0             0            0
global tx hash mapping          649,208    0.0    0.6      0            0
hash table column usage           1,169    0.3    0.5      5   32,482,472    5.0
i/o slave adaptor                     7    0.0             0            0
job workq parent latch                0                    0          136    3.7
job_queue_processes para            280    0.0             0            0
ksfv messages                 1,744,965    0.0             0           17    0.0
ktm global data                     199    0.0             0            0
kwqit: protect wakeup ti            239    0.0             0            0
lgwr LWN SCN                    295,394    0.0    0.1      0            0
library cache                55,951,057    1.4    0.4  44673      111,360    5.3
library cache load lock          15,232    0.0             0            0
library cache pin            29,345,261    0.2    0.3   3074            0
library cache pin alloca     21,024,601    0.2    0.5   3308            0
list of block allocation         34,233    0.0    1.0      2            0
loader state object free            442    0.0             0            0
longop free list parent          18,593    0.0             0        2,498    0.1
messages                        943,063    0.0    0.5     17            0
mostly latch-free SCN         3,957,459    0.1    0.2     57            0
```

```
msg queue latch                 1,232   68.1    0.1       0    1,714,717    3.0
multiblock read objects     7,538,869    0.3    0.3     440            0
ncodef allocation latch           141    0.0              0            0
object stats modificatio           19    0.0              0            0
post/wait queue               344,154    0.0    0.2       0      224,344    0.0
process allocation              3,326    0.0    1.0       0        3,325    0.0
```

cache buffers chains 等待出现了严重的问题。本来想今天开始静下心来写报告的，看来今天又有事做了。从现象上看，这是典型的 cache buffers chains 争用。一般来说，和应用程序有很大的关系。我首先找到了几个 Top SQL，会产生几百万甚至上千万的缓冲区获取。我打电话给万科长，希望开发厂商能够检查一下这几个 SQL,不过开发商那边的反馈是程序太复杂，可以优化，但是起码需要一个月的时间。我也研究了几个 SQL，都是超过 6 张表的连接，其中每个连接里起码有两三张超过千万条记录的大表，光是执行计划就有几十行，想要简单地调整一下索引或者调整一下表连接的顺序，很难达到优化效果。HCSP 系统从 6 月份进入业务高峰期，这个业务高峰要持续到八九月份。在业务高峰期间，业务量会比平时高近一倍。往年每年进入 6 月份，系统性能都会有所下降，而从今天的情况来看，系统几乎是不可用了。从各个维修站反馈回来的信息是，目前的系统查询一个订单，基本上需要 5 分钟左右，而之前只需要 10 秒左右。看样子通过应用进行优化的路子行不通，只能走其他的路子了。从等待事件看：

```
Event    waits      TimeOuts   Total
                               Wait Time(s)      Avg wait(ms)   waits/txn
-----------------------------------------------------------------------
         1,098,363  0
latch free                     606,375           552                  4.2
```

latch free 每次等待的时间为 552ms，等待 cache buffers chains 的时间是很不正常的。一般来说，cache buffers chains 等待是由于热块或者热链造成的。以前碰到的 cache buffers chains 问题大多数是热块引起的。不过热链引起的 cache buffers chains 闩锁争用也很常见。

Buffer Nowait 的比例是 99.93%，说明热块争用不是特别厉害：

```
    Number of  Cache  Buffer       Physical     Physical  Buffer  Complete   Busy
P   Buffers    Hit %  Gets         Reads        Writes    Waits   Waits      Waits
--- ---------- ------ ---------- ------------ --------- ------- --------- -----
D   595,500    94.9   739,848,992 37,571,278 817,573   0       0         619,303
K   43,670     100.0  84,408,249  55         1,035     0       0         1
-------------------------------------------------------------
```

125 分钟产生 62 万 buffer busy waits，虽然说比较多，不过还是可以接受的，毕竟是这么大的系统。从 cache buffers chains 闩锁的情况来看：

```
cache buffers chains kcbgtcr: kslbegin excl          0    227,599    414,454
cache buffers chains kcbrls: kslbegin                0    199,620     20,593
cache buffers chains kcbgtcr: fast path              0     73,320     77,189
cache buffers chains kcbzib: multi-block read:       0      7,494          0
cache buffers chains kcbzgb: scan from tail. no      0      5,609          0
cache buffers chains kcbzwb                          0      5,367     10,080
cache buffers chains kcbchg: kslbegin: bufs not      0      4,870      2,647
cache buffers chains kcbzib: finish free bufs        0      3,206      4,863
cache buffers chains kcbchg: kslbegin: call CR       0      2,407      1,455
```

```
cache buffers chains kcbgtcr: kslbegin shared           0      1,553    1,204
cache buffers chains     kcbgcur: kslbegin              0        884      910
cache buffers chains     kcbget: pin buffer             0        741    1,344
cache buffers chains     kcbgtcr                        0        702        1
cache buffers chains     kcbcge                         0        684       59
cache buffers chains     kcbget: exchange rls           0        667        2
cache buffers chains     kcbbic2                        0        267       24
cache buffers chains     kcbbxsv                        0        235      483
cache buffers chains     kcbnlc                         0        179      217
cache buffers chains     kcbget: exchange               0        158       12
```

kcbgtcr 和 kcbrls 占了绝大部分。从这些情况来看，热链的可能性比较大。由于某些热块存在大量的 CR 块，导致某些链相关的闩锁争用比较厉害。这种情况下，通过调整 DB Cache 的大小可以打散 hash 链。由于目前系统内存比较紧张，加大 DB Cache 的条件并不具备，而由于 default 池的命中率本身就不高（94.9%），减少 default 池的大小会加大 I/O 的负担，所以通过调整 DB Cache 的方法存在一定风险。现在来看，唯一的风险较小的办法是调整_DB_BLOCK_HASH_BUCKETS，从而打破热链，消除 latch free 等待。

晚上，通过 MSN 和美国的 John 以及澳大利亚的 Ben 进行了沟通，谈了我的想法。Ben 认为可以一试，但是解决问题的把握不大。John 认为我是在异想天开，让我慎重，还是要坚持优化部分应用。结束和 John 的谈话，关闭电脑时已经凌晨 2 点多了。该好好睡一觉了，明天一早还要赶到青岛去。

今日点评

今天青岛那边出现了 cache buffers chains 争用的问题。这个案例实际上是在 2004 年发生的，也是老白自认为比较得意的一个性能优化的案例。cache buffers chains 等待一般来说和应用的关系很大，当系统出现了大量的缓冲区获取时就容易出现。最好的方法是通过应用的调整来解决问题。而当时由于开发商和集成商之间存在的矛盾，导致当时无法很快对 SQL 进行优化，因此老白无奈之中，才想到去调整_DB_BLOCK_HASH_BUCKETS，打破热链。其实这种调整要基于对问题分析十分透彻的基础上，初入门的 DBA 千万不要轻易模仿。因为这种参数一旦调整失败，后果是十分严重的。

优化小技巧　STATSPACK 报告分析之 DB Cache1

DB Cache 的命中率在 STATSPACK 头能够很明确地看到。在 STATSPACK 中也包含了 DB Cache 的相关内容：

```
Buffer Pool Statistics for DB: HCSP  Instance: hcsp  Snaps: 111 -121
-> Standard block size Pools  D: default,  K: keep,  R: recycle
-> Default Pools for other block sizes: 2k, 4k, 8k, 16k, 32k

                                                           Free    Write  Buffer
     Number of Cache      Buffer    Physical    Physical Buffer Complete    Busy
P      Buffers Hit %        Gets       Reads      Writes  Waits    Waits   Waits
```

```
--- ---------- ----- ----------- ----------- ---------- ------- --------  ------
D      595,500  94.9 739,848,992  37,571,278    817,573       0        0 619,303
K       43,670 100.0  84,408,249          55      1,035       0        0       1
          -------------------------------------------------------------
```

实际上，通过命中率只能粗略了解 DB Cache 的情况。Buffer Pool Statistics 中包含了 DB Cache 中较为明细的信息，从中可以了解 DB Cache 的实际情况。首先是一个总体情况，每个缓冲池的命中率，缓冲区获取的数量、物理读的数量、物理写的数量、空闲缓冲区等待的数量、Write Complete Waits 的数量、Buffer Busy Waits 的数量。

如果系统中使用了多缓冲技术，那么每个缓冲区都会在这个报告中存在一行数据。D 代表 default 池，K 代表 keep 池，R 代表 recycle 池，另外还可能出现 2k、4k、8k、16k、32k 等，分别代表 *n*k 缓冲池。

在这些指标里，我们要注意几个指标。空闲缓冲区等待（Free Buffer Waits）用于统计进程申请 DB Cache 时，无法直接分配到空闲 Cache 的计数。对于一个正常状态的系统，这个值应该是 0。如果这个值非零，可能说明 DB Cache 不足。写完成等待（Write Complete Waits）事件是当某个进程申请 DB Cache 时，发现需要等待 DBWR 写入数据完成后才能获取到空闲的缓冲区。这个值如果不为零，那么说明 DB Cache 过小或者 DBWR 写入过慢，需要增加 DBWR 的数量。

Oracle 9i 开始引入了 DB Cache advisory，用于向 DBA 提供一个 DB Cache 的调整建议。下面是一个例子：

```
Buffer Pool Advisory for DB: HCSP  Instance: hcsp  End Snap: 121
-> Only rows with estimated physical reads >0 are displayed
-> ordered by Block Size, Buffers For Estimate

        Size for  Size      Buffers for  Est Physical          Estimated
P   Estimate (M) Factr         Estimate   Read Factor     Physical Reads
--- ------------ ----- ---------------- ------------- ------------------
K             32    .1            3,970     18,457.61        667,076,603
K             64    .2            7,940     10,200.68        368,662,788
K             96    .3           11,910      3,081.00        111,350,322
K            128    .4           15,880        422.90         15,284,067
K            160    .5           19,850         68.03          2,458,709
K            192    .5           23,820         18.48            667,727
K            224    .6           27,790          5.34            192,950
K            256    .7           31,760          1.54             55,505
K            288    .8           35,730          1.00             36,141
K            320    .9           39,700          1.00             36,141
K            352   1.0           43,670          1.00             36,141
K            384   1.1           47,640          1.00             36,141
K            416   1.2           51,610          1.00             36,141
K            448   1.3           55,580          1.00             36,141
K            480   1.4           59,550          1.00             36,141
D            480    .1           59,550          5.26     16,018,674,430
K            512   1.5           63,520          1.00             36,141
K            544   1.5           67,490          1.00             36,141
K            576   1.6           71,460          1.00             36,141
K            608   1.7           75,430          1.00             36,141
```

```
K           640   1.8        79,400          1.00           36,141
D           960    .2       119,100          3.36   10,245,019,912
D         1,440    .3       178,650          2.53    7,705,122,728
D         1,920    .4       238,200          2.09    6,359,203,109
D         2,400    .5       297,750          1.81    5,508,276,852
D         2,880    .6       357,300          1.60    4,884,851,052
D         3,360    .7       416,850          1.43    4,363,040,176
D         3,840    .8       476,400          1.28    3,899,800,284
D         4,320    .9       535,950          1.14    3,462,541,775
D         4,800   1.0       595,500          1.00    3,043,964,163
D         5,280   1.1       655,050          0.87    2,651,020,663
D         5,760   1.2       714,600          0.75    2,291,037,220
D         6,240   1.3       774,150          0.65    1,978,723,916
D         6,720   1.4       833,700          0.57    1,725,802,559
D         7,200   1.5       893,250          0.50    1,522,217,063
D         7,680   1.6       952,800          0.45    1,359,329,055
D         8,160   1.7     1,012,350          0.40    1,228,618,801
D         8,640   1.8     1,071,900          0.37    1,120,457,807
D         9,120   1.9     1,131,450          0.34    1,030,471,343
D         9,600   2.0     1,191,000          0.31      948,853,857
          -------------------------------------------------------------
```

这个表的字段的含义如下所示。

- P：缓冲池的名称。
- Size for Estimate (M)：估算的缓冲池大小。单位是 MB。
- Size Factor：其他估算值和当前大小之间的比例，当前缓冲池大小为 1。
- Buffers for Estimate：估算的缓冲池的数量（单位为 BLOCK）。
- Est Physical Read Factor：使用估算缓冲池的情况下，产生的物理读和当前实际值的百分比。当前缓冲池的物理读数量为 1。
- Estimated Physical Read：估算的缓冲池情况下物理读的大小。

据此，我们可以分析当缓冲池设置成某个大小时，对物理 I/O 产生的影响大概会有多大。这对我们在做优化时，判断数据块缓冲区的大小，具有十分好的帮助作用。

优化小技巧 STATSPACK 报告分析之 PGA

Process Global Area，也就是常说的 Program Global Area，简称 PGA，是数据库服务进程的私有空间。它包含全局变量数据结构和控制信息，比如一个游标的执行空间。当某个游标执行时，都需要在执行该游标的服务进程的 PGA 里分配一个运行空间。

调整 PGA 能够提高数据库的性能。总的来说，PGA 空间越大，能够得到越快的性能。拿排序来说，如果有足够的 PGA 排序区（sort area），那么所有的排序操作都可以在内存中完成，这样的操作是最优化的，称为 Optimal Execs。如果排序空间不足，那么就需要使用临时表空间将数据存储到临时段中。如果通过一次磁盘操作完成排序工作，称为 1-pass Execs。这种方式由于要进行磁盘操作，效率大大降低。如果通过一次磁盘操作还不能完成，需要进行多次磁盘交互，那么排序效率就十分低下了，这种情况称为 m-pass Execs。

Oracle 8i 只能通过调整*_area_size 参数来调整 PGA 的性能。Oracle 9i 有了革命性的改进，DBA 可以通过设置 PGA_AGGREGATE_TARGET 参数来限制 PGA 的大小，而由 Oracle 内部的智能算法来自动进行 PGA 各种工作缓冲区的分配工作。当 WORKAREA_SIZE_POLICY 设置为 AUTO 时，PGA 自动管理就会起作用（这两个参数都可以动态修改）。

要注意的是，在 Oracle 9i R2 版本中，PGA_AGGREGATE_TARGET 只能控制独立服务器模式的服务进程的工作区域。共享服务器模式的连接还是由*_area_size 来控制。Oracle 10g 可以控制独立服务器和共享服务器模式的所有工作区域。

一般情况下，如果使用裸设备，那么对于 OLTP 系统，PGA_AGGREGATE_TARGET 用以下公式来分配：

```
PGA_AGGREGATE_TARGET  = (<Total Physical Memory > * 80%) * 20%
```

对于 OLAP 系统，可以按照下面的公式计算：

```
PGA_AGGREGATE_TARGET  = (<Total Physical Memory > * 80%) * 50%
```

要注意的是，以上只是一个建议的参数设置。如果要知道系统中设置什么样的值比较合理，建议通过 STATSPACK 报告来进行分析。具体的分析方法请参照本章的相关章节。通过部分系统视图也可以对 PGA 的使用情况进行分析，得到相应的优化建议。

通过 v$pgastat 视图可以得到 PGA 的详细情况：

```
SELECT * FROM v$pgastat;
NAME                                                VALUE
---------------------------------------------------------
aggregate PGA target parameter                      524288000 bytes
aggregate PGA auto target                           463435776 bytes
global memory bound                                 25600 bytes
total PGA inuse                                     9353216 bytes
total PGA allocated                                 73516032 bytes
maximum PGA allocated                               698371072 bytes
total PGA used for auto workareas                   0 bytes
maximum PGA used for auto workareas                 560744448 bytes
total PGA used for manual workareas                 0 bytes
maximum PGA used for manual workareas               0 bytes
over allocation count                               0 bytes
total bytes processed                               4.0072E+10 bytes
total extra bytes read/written                      3.1517E+10 bytes
cache hit percentage                                55.97 percent
```

其中主要指标的含义如下所示。

- aggregate PGA auto target：Oracle 可以用于 PGA 工作区的空间。一般比参数略小。
- total PGA used for auto workarea：系统实际使用的 PGA 空间。
- maximum PGA used for auto workareas：上次数据库启动后 PGA 空间达到的最大值。
- total PGA inuse：当前正在使用的 PGA。可以从 v$process 的 pga_used_mem 字段中获取。

以下是 Oracle 9i R2 以后的版本具有的指标。

- over allocation count：如果 PGA_AGGREGATE_TARGET 太小，动态扩展 PGA 空间的次数。如果系统的物理内存足够，那么应该设置一个足够大小的 PGA_AGGREGATE_TARGET，使该统计值为 0。
- cache hit percentage：PGA 工作区使用内存的百分比。如果内存足够，那么这个值应该接近或者等于 100%。

v$sql_workarea_histogram 视图中存放了 PGA 的使用情况。通过该视图可以了解 PGA 工作区域的使用情况。下面的语句可以检查 PGA 的工作情况：

```
SELECT LOW_OPTIMAL_SIZE/1024 low_kb,(HIGH_OPTIMAL_SIZE+1)/1024 high_kb,
       optimal_executions, onepass_executions, multipasses_executions
FROM   v$sql_workarea_histogram
WHERE  total_executions != 0;
LOW_KB HIGH_KB OPTIMAL_EXECUTIONS ONEPASS_EXECUTIONS MULTIPASSES_EXECUTIONS
------ ------- ------------------ ------------------ ----------------------
8      16      156255             0                   0
16     32      150                0                   0
32     64      89                 0                   0
64     128     13                 0                   0
128    256     60                 0                   0
256    512     8                  0                   0
512    1024    657                0                   0
1024   2048    551                16                  0
2048   4096    538                26                  0
4096   8192    243                28                  0
8192   16384   137                35                  0
16384  32768   45                 107                 0
32768  65536   0                  153                 0
65536  131072  0                  73                  0
131072 262144  0                  44                  0
262144 524288  0                  22                  0
```

这个视图可以显示各种工作区的使用情况。OPTIMAL_EXECUTIONS 是不需要通过临时表空间交换的操作次数。ONEPASS_EXECUTIONS 是需要进行一次临时表空间交换的操作次数。MULTIPASSES_EXECUTIONS 是需要进行多次临时表空间交换的操作次数。原则上，最佳选择是 100%的操作都是内存操作。如果内存不足，某些大型操作是 1-pass 的也可以接受。出现 m-pass 的操作会引起性能的急剧下降。

v$sql_workarea_active 视图显示当前工作区的活动情况。小于 64KB 的小型排序不会被统计。通过该视图可以监控目前系统的工作区使用情况。比如：

```
SELECT to_number(decode(SID, 65535, NULL, SID)) sid,
       operation_type OPERATION,trunc(EXPECTED_SIZE/1024) ESIZE,
       trunc(ACTUAL_MEM_USED/1024) MEM, trunc(MAX_MEM_USED/1024) "MAX MEM",
       NUMBER_PASSES PASS, trunc(TEMPSEG_SIZE/1024) TSIZE
FROM v$sql_workarea_active
ORDER BY 1,2;
```

```
SID OPERATION          ESIZE     MEM       MAX MEM    PASS TSIZE
--- ----------------- --------- --------- --------- ----- -------
8   GROUP BY (SORT)   315       280       904         0
8   HASH-JOIN         2995      2377      2430        1   20000
9   GROUP BY (SORT)   34300     22688     22688       0
11  HASH-JOIN         18044     54482     54482       0
12  HASH-JOIN         18044     11406     21406       1   120000
```

下面介绍一下这些数据的具体含义。

- SID：产生排序操作的会话 ID。
- OPERATION：操作的类型。
- ESIZE：PGA 管理器估计的本次操作需要的总内存数量（单位是 KB）。
- MEM：目前正在使用的内存数量（单位是 KB）。
- MAX MEM：该会话曾经使用的最大 PGA 内存数量（单位是 KB）。
- PASS：0 代表内存排序，1 代表 1-pass 排序，m 代表 m-pass 排序。
- TSIZE：转储到临时表空间的大小（单位是 KB）。

对于 Oracle 9i R2 版本以后的数据库，还提供了下面两个新视图。

- v$pga_target_advice：提供 PGA 使用的建议数据。
- v$pga_target_advice_histogram：PGA 建议直方图。如果要调整 PGA 的大小，这个视图十分重要。

对于一个正在运行的实际系统，通过这两个视图可以了解如何配置 PGA_AGGREGATE_TARGET 是比较合适的。要使用这两个视图，首先要设置 statistics_level 为 typical 或者 all。如果该参数设置为 basic，那么这两个视图不会提供任何数据。v$pga_target_advice 视图提供 PGA_AGGREGATE_TARGET 和 PGA 命中率以及 PGA Overalloc 的数量。比如：

```
SELECT round(PGA_TARGET_FOR_ESTIMATE/1024/1024) target_mb,
       ESTD_PGA_CACHE_HIT_PERCENTAGE cache_hit_perc,
       ESTD_OVERALLOC_COUNT
FROM   v$pga_target_advice;
TARGET_MB  CACHE_HIT_PERC ESTD_OVERALLOC_COUNT
---------- -------------- --------------------
63         23             367
125        24             30
250        30             3
375        39             0
500        58             0
600        59             0
700        59             0
800        60             0
900        60             0
1000       61             0
1500       67             0
2000       76             0
3000       83             0
4000       85             0
```

在设置 PGA_AGGREGATE_TARGET 时，尽量选择物理内存可以支撑的、命中率比较高，并且 ESTD_OVERALLOC_COUNT 为 0 的建议值。在上面的例子里，PGA_AGGREGATE_TARGET 取值的最小值是 375，低于这个值，OVERALLOC 是非零的。如果物理内存充足，设置为 4 000MB 或者更高可以提高 PGA 的性能。

v$pga_target_advice_histogram 视图提供 PGA 使用情况的直方图情况。通过这个统计直方图，可以了解系统中每个规模的区域的使用情况。如果某类较小的工作区域产生了 m-pass 操作，那么说明 PGA 的设置偏小。

和 Oracle 8i 比较，Oracle 9i 的工作区管理更加灵活。Oracle 9i 提供动态工作区管理的能力。在 Oracle 8i 下，分配的工作区域将不会归还给操作系统，会导致 Oracle 对物理内存的过度占用，严重时会导致系统产生换页操作。Oracle 9i 的工作区用完后会自动归还，这样减轻了数据库服务程序对物理内存的占用。

如果你不习惯通过 SQL 脚本去分析，那么完全可以使用 STATSPACK 报告。因为这些信息在 STATSPACK 报告的 PGA 分析章节均有十分明确的数据。



青　　岛

为了尽快解决万科长那边的问题，我赶一大早的飞机去青岛，不过飞机有点晚点，到客户那已经是中午了。我出发前就给万科长发了个邮件，说明我准备调整的参数，希望他们能够申请一个临时停机，时间就是中午的 12：30～13：30。这样趁着中午时间可以重启服务器。在飞机上我把万科长发来的数据整理了一下，发现系统这几天的负载比前几天要高一些，和 14 日以前的数据相比：

- REDO SIZE 增加 30%；
- 逻辑读增加 90%；
- 块变更增加 35%；
- 物理读增加 160%；
- 调用次数增加 35%；
- 解析增加 42%；
- 硬解析增加 20%；
- 排序增加 48%；
- 执行增加 42%。

到了现场，我检查了一下系统，虽然是中午，CPU 的使用率依然是 100%。正是由于业务量的突然增加，导致对缓冲区的访问猛增，最终由于闩锁的争用消耗了大量的 CPU 资源。通过分析，我发现：

```
addr=C00000018DB91268 sleeps=2501
XXXX.CD_ROAD 41 25 18317
XXXX.T_PRODTYPE 4 25 1838
XXXX.CD_POS 86 25 1467
XXXX.CD_ROAD_FK5 13 32 1048
XXXX.CD_SERVE_AREA 116 25 75
XXXX.WO_SERVICE_INFO_N24 4 15500 56
addr=C00000018DB90FD8 sleeps=2447
XXXX.CD_ROAD 41 23 20264
XXXX.CD_POS 86 23 1805
XXXX.T_PRODTYPE 4 23 1729
XXXX.CD_ROAD_FK5 13 30 1044
XXXX.CD_SERVE_AREA 116 23 639
XXXX.WO_SERVICE_INFO_N24 4 15498 76
XXXX.WO_SERVICE_INFO_N24 5 6436 65
```

```
addr=C00000018DB918D0 sleeps=2098
XXXX.CD_ROAD 41 30 19120
XXXX.T_PRODTYPE 4 30 4477
XXXX.CD_POS 86 30 1284
XXXX.CD_SERVE_AREA 116 30 174
XXXX.WO_SERVICE_INFO_N24 5 6443 88
addr=C00000018DB91120 sleeps=1910
XXXX.CD_ROAD 41 24 18487
XXXX.CD_POS 86 24 1471
XXXX.CD_ROAD_FK5 13 31 1057
XXXX.T_PRODTYPE 4 24 926
XXXX.CD_SERVE_AREA 116 24 86
XXXX.WO_SERVICE_INFO_N22 1 328 57
XXXX.WO_SERVICE_INFO 0 968 54
addr=C00000018DB914F8 sleeps=1753
XXXX.CD_ROAD 41 27 18505
XXXX.T_PRODTYPE 4 27 3797
XXXX.CD_POS 86 27 1291
XXXX.CD_SERVE_AREA 116 27 372
XXXX.WO_SERVICE_INFO_N24 4 15502 50
addr=C00000018DB913B0 sleeps=1717
XXXX.CD_ROAD 41 26 19920
XXXX.T_PRODTYPE 4 26 4951
XXXX.CD_POS 86 26 1275
XXXX.CD_SERVE_AREA 116 26 949
XXXX.WO_SERVICE_INFO_N24 4 15501 67
XXXX.WO_SERVICE_INFO 0 970 45
XXXX.CD_SERVE_AREA_FK3 12 1 45
addr=C00000018DB90E90 sleeps=1661
XXXX.CD_ROAD 41 22 20753
XXXX.T_PRODTYPE 4 22 4177
XXXX.CD_POS 86 22 1467
XXXX.CD_SERVE_AREA 116 22 1241
XXXX.CD_ROAD_FK5 13 29 1049
XXXX.CD_SERVE_AREA_FK4 45 29 65
XXXX.WO_SERVICE_INFO_N24 5 6435 42
addr=C00000018DB91788 sleeps=1503
XXXX.CD_ROAD 41 29 20632
XXXX.T_PRODTYPE 4 29 2537
XXXX.CD_POS 86 29 1281
XXXX.CD_SERVE_AREA 116 29 339
XXXX.WO_SERVICE_INFO_N24 5 6442 47
XXXX.WO_FM_REPLACE 0 248 34
XXXX.WO_SERVICE_INFO 0 2140 28
XXXX.WO_SERVICE_INFO_N22 1 333 17
XXXX.SP_TRAN 60 222 17
```

从中可以看出，热点十分集中。主要集中在 CD_ROAD、T_PRODTYPE 和 CD_POS 相关的数据块上。这几张表都是变更十分频繁的，存在大量的 CR 块。

离申请的停机时间越来越近了，我已经没有时间做过多分析了，并且系统已经基本不可用，所以我只能冒险先做一下调整了。我和万科长说只能先试试，机会有六成。万科长说没关系，就是有一成也要试，死马当活马医吧。对于 Oracle 9.2，_DB_BLOCK_HASH_BUCKETS 被设置为大于

2*缓冲区数量的素数。我使用 Delphi 写了一个小程序，计算出比当前_DB_BLOCK_HASH_BUCKETS 值略大的一个素数。将该参数设置为这个素数，修改后重启数据库。正好是下午 1：30 了，业务高峰会出现在 3：00 以后，1：30 的业务量也应该相当于上午 9：00 左右的情况。而今天上午 9 点多的时候系统几乎是不可用的，因此这个参数调整后有没有作用，重启业务后很快就可以看出来。应用重启后，我们紧张地盯着 OEM 的监控界面。活动会话数量一直维持在 150 左右，平均事务响应时间在 2s～3s，这已经比上午的超过 7s 的平均事务响应时间好了很多。不过这会不会是由于系统刚刚恢复，用户还没有开始使用有关呢？

万科长他们在往各地的工贸公司打电话咨询目前的系统情况，反馈回来的信息都还不错。打开 QQ，在 QQ 群上咨询，反馈的结果是比上午好多了，现在使用基本正常。从监控界面上看，平均每秒的事务数已经超过 40 了，这比上午平均每秒 20～30 个事务已经高出很多，活跃会话数量一直保持在 100～150 之间的正常范围。平均事务响应时间也在 3s 以内。CPU 使用率也基本保持在 60%左右。我和万科长说干脆我们等 3：00 时看看高峰期的情况再去吃饭，反正现在大家去吃饭也没什么胃口。于是大家就开始了焦急的等待。

已经是下午 3：10 了，系统好像一切正常。万科长他们也已经咨询了 6 个工贸公司，40 多个网点，反馈回来的结果是一切正常，看样子调整起到了效果。到这个时候大家才感觉肚子有点饿了，到现在为止还没有吃中午饭。大家一起步行 10 多分钟，找到一家韩国餐馆，点了几个石锅拌饭，大吃了起来。吃饭时，万科长还不停地拿出手机看看，是不是有电话，我开玩笑说：“万科长，你的手机看样子坏了，怎么一个投诉电话都没有？”看样子他的心里还是有一点不踏实。仅仅调整了一个参数，就真的能解决这么严重的性能问题吗？

今天下午系统一直工作正常，到了下午 6：00 时，系统负载明显下降了。CPU 使用率下降到 30%了，平均事务响应时间也提高到了 1 秒多。看样子今天的调整是成功的。我准备赶晚班飞机回沈阳，万科长说别急，晚上给你接风。我说：“万科长，你是怕明天还有事不敢放我走吧。”万科长说，“两者都有，一是表示感谢，二是你多待一天，看看明天上午的情况，我心里踏实。”和老于联系了一下，沈阳一切正常，我晚回去一天也关系不大。看样子晚上可以和万科长好好喝一下青岛著名的青啤原浆了。

晚上和万科长他们在啤酒街吃饭。啤酒街就在青啤旁边的一个小巷子里，现在成为一条酒吧街，每天晚上人山人海的。不过到啤酒街喝酒的大多数都是外地游客，青岛本地人喜欢喝啤酒，不过大多数人都喜欢喝散装的生啤，在青岛街头到处都能看到卖生啤的车，和其他地方不同的是，青岛这边卖生啤的是用塑料袋装啤酒。每天傍晚，在青岛街头到处可以看到拎着一袋袋啤酒的人。

啤酒街的人很多，好不容易找到一张空桌子。大家坐下来，点了几个小海鲜，要了几扎青岛原浆。今天问题解决了，所以大家心情都很放松。万科长这几天一直如坐针毡，被工贸公司投诉得抬不起头来，今天一旦压力释放，酒就喝得很畅快。半扎啤酒下肚，万科长的话也多了起来。在他这个位置上，确实也是压力很大。这么大一个系统，全国 8 000 多个客户端使用，而服务器仅仅是一台 8 颗 CPU、16GB 内存的 RP 7420，系统资源的紧张可想而知。虽然开发商已经尽了最大的努力，系统还是经常出现一些状况。我今天下午初步看了看系统的情况，感觉在底层架构

以及物理数据字典设计方面还存在一些问题，这些隐患不除，今后还时不时会出现一些问题。我和万科长说了我的担忧，并说如果万科长能申请一笔费用，做一个整体调优，可以确保两年内不会出现严重的性能问题，否则每隔 3 个月系统会出现一次周期性的性能问题，如果正好碰到业务量比较大的时候，就很可能出现类似这几天的性能危机。

晚上回到酒店，上网时就收到了 Ben 发来的一个邮件。正文什么都没有，只有一个附件，内容是 Rich Niemiec 的 *Advanced Tuning with STATSPACK*。这份文档全面地讲述了通过 STATSPACK 报告分析 latch free 问题的一些基本方法。看了一下，感觉收获良多。本来想在网上和 Ben 聊聊，一看表，已经是晚上 10 点多了，Ben 那边已经是 12 点多了。不过 John 这个时候应该在公司。我打开雅虎通（Yahoo Messenger），发现 John 正好在线，就把今天调整的结果告诉了 John，John 感觉很兴奋。他也研究过 cache buffers chains 闩锁争用的问题，以前也尝试过调整 BUCKETS 的数量来解决争用问题，不过没有成功。这个案例也是他听说的第一个通过调整 BUCKETS 参数获得成功的例子，值得好好总结一下。我把所有资料都发给他，希望他有时间再好好研究一下，看看能不能从理论上来更好地解释这个案例。我看了一下下午采集的 STATSPACK 报告，确实是太神奇了。

```
Load Profile
~~~~~~~~~~~~                            Per Second       Per Transaction
                                   ---------------       ---------------
                  Redo size:            258,351.03              5,729.49
              Logical reads:             95,681.57              2,121.95
              Block changes:              1,634.66                 36.25
             Physical reads:              5,036.17                111.69
            Physical writes:                186.52                  4.14
                 User calls:              3,663.65                 81.25
                     Parses:                614.02                 13.62
                Hard parses:                 12.09                  0.27
                      Sorts:                 31.67                  0.70
                     Logons:                  0.46                  0.01
                   Executes:                868.70                 19.27
               Transactions:                 45.09

  % Blocks changed per Read:    1.71    Recursive Call %:          26.34
 Rollback per transaction %:    0.47       Rows per Sort:        2659.06

Instance Efficiency Percentages (Target 100%)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
            Buffer Nowait %:   99.71       Redo NoWait %:         100.00
            Buffer  Hit   %:   94.76    In-memory Sort %:          99.99
            Library Hit   %:   99.40        Soft Parse %:          98.03
         Execute to Parse %:   29.32         Latch Hit %:          99.84
Parse CPU to Parse Elapsd %:    8.50     % Non-Parse CPU:          95.29

 Shared Pool Statistics        Begin   End
                               ------  ------
             Memory Usage %:   80.86   81.12
    % SQL with executions>1:   87.88   85.32
  % Memory for SQL w/exec>1:   98.99   99.16
```

```
Top 5 Timed Events
~~~~~~~~~~~~~~~~~~                                                         % Total
Event                                          Waits        Time (s) Ela Time
-------------------------------------------- ------------ ----------- --------
db file sequential read                        20,137,988   1,071,848    74.05
buffer busy waits                               2,245,704     193,339    13.36
db file scattered read                          2,139,401     124,182     8.58
CPU time                                                       44,780     3.09
io done                                           917,807       5,398      .37
          -------------------------------------------------------------
```

db file sequential read 等待占了绝大多数，并且每次等待时间高达 50ms。I/O 又出现了一定的瓶颈，但是 CPU 使用率明显下降了，平均事务响应时间也下降到 2 秒多了。I/O 问题是这个系统的老问题，实际上这个案例的目的是解决 CPU 瓶颈问题。latch free 和 CPU 瓶颈是互为影响的，由于 latch free，闩锁获取消耗了大量的 CPU 资源，从而导致 CPU 出现瓶颈，反过来 CPU 不足更加加剧了闩锁的性能问题。当 CPU 出现瓶颈时，I/O 问题不明显了，甚至 I/O 性能恢复了正常。通过恶化 I/O 来换取 CPU 负载的减轻是调优时经常使用的技术。因此我在确定调整策略时，坚持不加大 DB Cache。因为加大 DB Cache 会增加 CPU 消耗，对于解决目前的问题不利。因为给我做优化的机会只有一次，所以我的优化目的不是使系统达到最优化，而是解决目前的问题，使系统恢复正常运行状态。只要客户能够接受就可以了，而不是去解决所有的问题。因为优化是一个十分复杂的系统工程，就像解多元多次方程组，变元之多，结果之不确定是很多人想象不到的。因此，每次优化我都会以目标为导向，而不是以技术为导向。一旦实现目标就收手，而不会去做画蛇添足的事情。

今日点评

今天老白通过调整隐含参数达到了优化目的。不过在实际工作中隐含参数的使用要十分谨慎。隐含参数一般用于改变数据库的一些工作特性。在 Oracle 官方发布的文档中，很少有隐含参数的介绍。在 Metalink 上的一些仅限内部阅读（Oracle 内部使用）的文档中，包含了对隐含参数使用的描述。在 Oracle 粉丝网上有一个专门的栏目用于大家发布隐含参数相关的文档，大家也可以去那里查阅。通过下列脚本，可以查询某个隐含参数的取值：

```
set line 132
col "session value" format a40 truncate
col "parameter" format a40 truncate

SELECT
   a.ksppinm  "Parameter",
   a.ksppdesc "Description",
   b.ksppstvl "Session Value"
   c.ksppstvl "Instance Value"
FROM
   x$ksppi a,
```

```
    x$ksppcv b,
    x$ksppsv c
WHERE
    a.indx = b.indx
    AND
    a.indx = c.indx
    AND
    a.ksppinm = lower('&1');
```

要注意的是，因为隐含参数的改变可能影响到数据库的运行状态，所以调整隐含参数一定要经过资深人员的确认或者在 Metalink 上开一个服务请求，向 Oracle GCS 的技术人员咨询。

优化小技巧 如何优化闩锁

闩锁竞争是数据库优化工作中经常碰到的问题，会极大影响数据库系统的性能。闩锁竞争的主要表现为系统响应时间急剧增加，系统 CPU 使用率大幅度增加，严重时会出现 CPU 长时间处于 100%的状态。通过 STATSPACK 报告可以看出，latch free 等待成为 Top 5 等待中的一个。

闩锁竞争会导致大量的系统等待。由于闩锁的机制是重试加休眠的方式，所以如果存在大量闩锁竞争，系统性能将急剧下降，并且会导致系统出现十分恶劣的性能问题。解决闩锁竞争问题是数据库系统调优中最具挑战性的工作。闩锁是一种开销比较小的短时间锁定资源的方式，通过设置锁定位来实现。latch free 等待的含义是当某个操作需要获得某个资源时其他会话已经锁定了该资源，导致 latch free 等待。闩锁竞争问题分为独立闩锁竞争和系统级闩锁竞争。独立闩锁竞争是由于对每个资源的竞争访问引起的，而系统级闩锁竞争是由于系统中大量访问热点资源引起的。对于这两种竞争，有不同的处理方法。

要讨论独立闩锁竞争，首先要介绍几个关键参数（为了便于下面的讨论，对每个参数都起了别名）。

- 闩锁地址：这里指 P1 参数，类型为 RAW。（注意：对于 32 位系统和 64 位系统，以及不同的 Oracle 版本，长度略有不同。）
- 闩锁号：`v$latchname` 中的 `latch#`值。这里指 P2 参数。
- tries：为了获取该闩锁，进行了多少次尝试。这里指 P3 参数。
- 等待时间：如果一个会话等待一个闩锁，那么首先进行短暂的休眠，然后重新测试这个闩锁，如果还是没有获取该闩锁，那么增加 P3，会话重新进入等待。通过闩锁地址可以找到该闩锁的相关情况：

```
SELECT name, 'Child '||child#, gets, misses, sleeps
  FROM v$latch_children
 WHERE addr='&P1'
UNION
SELECT name, null, gets, misses, sleeps
  FROM v$latch
 WHERE addr='&P1';
```

通过闩锁号可以直接从 `v$latchname` 中查找闩锁的信息：

```
SELECT * FROM v$latchname
   WHERE latch# = &P2;
```

闩锁等待是十分短的等待操作。如果系统中的闩锁等待时间比较长，那么就需要查清哪些闩锁的竞争十分激烈。使用 STATSPACK 报告或者 BSTAT/ESTAT 报告中的相关信息，可以查找闩锁等待的情况。生成这些报告信息的基础是 v$latch。下面的语句可以查找竞争比较多的闩锁：

```
SELECT latch#, name, gets, misses, sleeps
    FROM v$latch
    WHERE sleeps>0
ORDER BY sleeps  DESC;
```

在这个列表的最前面列出的就是情况最糟糕的闩锁。通过上面的语句进行进一步的分析，可以进一步定位闩锁竞争。下一步的定位需要使用另外一个视图 v$latch_children（这个视图是 Oracle 7.3 版本以后才有的）：

```
SELECT addr, latch#, gets, misses, sleeps
    FROM v$latch_children
    WHERE sleeps>0
    and latch# = &LATCH_NUMBER
ORDER BY sleeps ;
```

这里的 LATCH_NUMBER 指的是上面查询中找到的 latch#。这里列出第一个查询中闩锁的子闩锁的情况。如果这个查询没有返回值，那么说明这个闩锁没有子闩锁。

如果返回的结果有多行，那么最重要的是找出 sleeps 比较大的闩锁，或者找出占 sleeps 等待最大比例的几个闩锁。找到这些闩锁后，可以通过查询 v$latch_holder 来找到这些闩锁的持有者。

闩锁竞争的问题解决起来十分复杂，没有绝对的方法可以定位和解决闩锁竞争的问题。如果找不到特定的闩锁，所有的闩锁都存在等待问题，那么需要检查 CPU 资源是否充足。如果 STATSPACK 报告中，latch free 处于主要事件的前几位，那么说明系统存在闩锁竞争问题。在报告中，任何 Miss 或者 No Wait Miss 超过 1%的闩锁都需要我们详细分析，看看是否存在问题。对于一些比较常见的闩锁问题，我们将在下面详细讨论。

1. 共享池

大多数共享池的操作都需要使用共享池闩锁，比如说共享池中对象的分配、释放和查找。过多地使用文字化的 SQL 语句（没有使用绑定变量）会导致 SQL 分析数量的增长，而 SQL 分析需要多次使用**共享池**闩锁，从而导致**共享池**闩锁等待。另外，如果共享池的容量不足，也会导致共享池中的对象产生抖动现象，从而加剧该闩锁的竞争。

2. 库缓存

从 Oracle 7.2 开始，**库缓存**闩锁包含子闩锁。对库缓存的分配、释放和查找等操作需要使用该闩锁。大量使用文字化的 SQL 或者共享池不足会导致**库缓存**的变更加剧，甚至产生**库缓存** 的抖动，从而加剧该闩锁的竞争。

3. 缓存缓冲区 LRU 链

缓存缓冲区 LRU 链闩锁是管理 DB Cache 的 LRU 队列。从 Oracle 7.3 开始，数据库支持多个

缓存缓冲区 LRU 链闩锁。通过调整 DB_BLOCK_LRU_LATCHES 参数，可以调整这个闩锁的数量。要注意的是，每个数据库的版本对于调整该参数的规则都各不相同，因此在调整这个参数时要参考相关版本的参考手册。

这个闩锁的主要竞争原因是过多的全表扫描或者过多地扫描选择性差的索引。对于 Oracle 7 或者 Oracle 8.0，如果没有使用 SORT_DIRECT_WRITES，大量的排序操作也会导致该闩锁的竞争。

4. 缓存缓冲区链

产生**缓存 缓冲区链**的竞争的原因是热块冲突或者热链冲突。热块冲突很多 DBA 都经常遇到，就是多个会话同时访问一个块，导致竞争。热块竞争的原因比较复杂，下面简要列出几个。

- I/O 性能问题会导致热块冲突。
- 应用设计缺陷会导致过多地对同一数据进行并发访问，从而导致热块冲突。对于此类问题，使用较小的块大小或者使用散列分区等技术可以减少热块冲突。
- 某些通过序列号产生的主健或者索引会导致对索引访问的热块冲突。对于此类问题，使用反转键索引可以解决热块冲突问题。

缓存缓冲区链冲突的另外一种情况是热链冲突。也就是说，对同一个 hash 链的并发访问量过大，导致该闩锁竞争。从 Oracle 8i 开始，支持很大的 DB Cache，因此热链冲突也比较容易发生。如果是热链冲突，调整_DB_BLOCK_HASH_ BUCKETS（这里以 Oracle 9i 为例，Oracle 的各个版本略有不同）可以改变散列链的分布，有助于解决**缓存缓冲区链**等待问题。

对于 Oracle 8，下面的 SQL 可以查找到产生该闩锁冲突的数据块：

```
SELECT File# , dbablk, class, state
FROM x$bh
WHERE hladdr='&ADDR_OF_CHILD_LATCH' ;
```

对于 Oracle 7，x$bh 的结构略有不同，需要使用如下 SQL 查询：

```
SELECT dbafil FILE# , dbablk, class, state
FROM x$bh
WHERE hladdr='&ADDR_OF_CHILD_LATCH';
```

如果查询结果只有几条记录，说明这几个数据块是十分热的热块。通过反复观察 x$bh，可以定位比较热的块。对于 Oracle 8i 数据库来说，由于 cache buffer chains 闩锁的数量较少，如果 DB Cache 比较大，那么一个闩锁管理的缓冲区数量较大。这些缓冲区如果比较“热”，那么也会出现较严重的闩锁竞争，此时调整闩锁的数量可以解决部分问题。

下面的 SQL 语句可以帮助 DBA 检查闩锁的情况：

```
SELECT "WHERE", SLEEP_COUNT, WTR_SLP_COUNT, LONGHOLD_COUNT
    FROM v$latch_misses
   WHERE parent_name='&ADDR_OF_PROBLEM_LATCH'
   ORDER BY ;
```

闩锁竞争会导致 CPU 资源被过度消耗，从而导致系统性能严重下降。反过来，如果 CPU 资源不足，那么也有可能造成较为严重的闩锁竞争。在实际工作中，需要我们识别出到底哪个是因哪个是果。

6月16日 SQL 优化效果

昨天刚调整了系统，所以今天一大早我就起来了，这也是我多年给客户服务养成的习惯。一般来说实施调整后的第二天，我会在客户上班前赶到现场，以防止出现异常现象。一大早打了辆车，拿着行李到老万那里去看看，如果现场没什么问题，我就准备直接去机场了。走进维护部的办公室，发现气氛比昨天这时候要轻松得多。万科长正在打电话，看到我进来，做了个表示歉意的手势。宋工连忙招呼我坐下，顺手从旁边的纸箱里拿出一瓶矿泉水递了过来。宋工是万科长手下的 DBA，在这个项目中我们已经配合多次，是个工作十分认真的年轻人。

我拿出笔记本电脑，连上系统，使用 OEM 查看了一下系统状态。现在是早上 8：30，各地的工贸公司和服务网点陆续上班了，不过业务高峰要 9：00 以后才会到来。从现在的情况看，数据库运行十分正常，平均事务响应时间稳定在 2～3 秒之间，看样子昨天的调整确实起到了作用。万科长打了半天电话才走过来，说：“这么急着走啊，休息一天，晚上一起喝点，给你庆功。”我说：“下回吧，沈阳那边事情也很急，今天必须赶回去。看样子这边不会有什么问题了，你请我喝酒是怕系统有什么反复吧。”万科长连忙解释，我说这只是玩笑而已，不要当真，确实沈阳那边事情很紧急，必须马上回去。

9：30 了，系统的状态一直维持在刚才那种情况，latch free 等待没有再次出现，看样子系统真的没问题了。和万科长寒暄了几句后，我就告辞了。如果运气好，我还赶得上中午那班飞机。看样子今天运气也不错，和万科长道别后一出工贸公司的大门，就看到一辆出租车正准备掉头。上了出租车，马上打电话订机票。青岛到沈阳的航班不多，如果赶不上中午的航班，就只能晚上走了。订好机票后，才发现是个老司机，大概有 60 多岁了，师傅虽然不年轻了，不过车开得飞快。我说：“师傅，不着急，我不赶时间。”师傅说：“你是不赶时间啊，我还要赶时间挣钱呢。”山东这地方人豪爽，火气也旺，碰到这么个老师傅，也真没办法。师傅开得够快，不到 10：00 就到机场了。办理登机牌时，工作人员告诉我飞机晚点，具体晚多少现在还不清楚，刚才的好心情顿时烟消云散了，今天可能又要在机场吃午饭了。还好有贵宾卡，只好在贵宾室里找个有网线和电源线的位置，坐下来上网。

端了杯茶，随手又拿了几块饼干，刚刚坐下，电话就响了。电话是一个客户打过来的，说是碰到一个很奇怪的问题。在一张上千万条记录的大表里，做一个 SELECT * FROM <TAB_NAME> WHERE ROWNUM<100，居然 10 多秒钟才出来。我问他这张表是不是碎片很厉害，他说不可能有碎片，因为这张表是昨天晚上才 IMP 进去的，昨天还没问题，今天就出问题了。而且这张是话

单表，不可能会做删除操作的，不会有碎片。

我让他马上做个 10046 的 Trace 发过来。10 分钟后，他通过 QQ 把 Trace 文件发过来了：

```
SELECT *
FROM
ttt where rownum<100
call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.14       0.17         44        198          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        8      3.71       5.86      67489      68340          0          99
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total       10      3.85       6.03      67533      68538          0          99
```

从上面看，确实产生了 67 533 个物理读和 68 538 个逻辑读，执行时间为 6.03s。从等待事件来看：

```
BINDS #39:
EXEC #39:c=0,e=88,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=1,tim=1422207486718
WAIT #39: nam='SQL*Net message to client' ela= 7 driver id=1650815232 #bytes=1 p3=0
obj#=206418 tim=1422207486810
WAIT #39: nam='SQL*Net more data to client' ela= 203 driver id=1650815232 #bytes=2002
p3=0 obj#=206418 tim=1422207487071
WAIT #39: nam='SQL*Net more data to client' ela= 66 driver id=1650815232 #bytes=2020
p3=0 obj#=206418 tim=1422207487175
WAIT #39: nam='db file scattered read' ela= 515 file#=146 block#=92900 blocks=5
obj#=206418 tim=1422207488208
WAIT #39: nam='db file scattered read' ela= 918 file#=146 block#=92905 blocks=8
obj#=206418 tim=1422207489579
WAIT #39: nam='db file scattered read' ela= 2121 file#=146 block#=92914 blocks=7
obj#=206418 tim=1422207492091
WAIT #39: nam='db file scattered read' ela= 617 file#=146 block#=92921 blocks=8
obj#=206418 tim=1422207493135
WAIT #39: nam='db file scattered read' ela= 493 file#=146 block#=92930 blocks=7
obj#=206418 tim=1422207494016
WAIT #39: nam='db file scattered read' ela= 1666 file#=147 block#=897417 blocks=8
obj#=206418 tim=1422207496049
WAIT #39: nam='db file scattered read' ela= 1026 file#=147 block#=897426 blocks=7
obj#=206418 tim=1422207497350
WAIT #39: nam='db file scattered read' ela= 378 file#=147 block#=897433 blocks=8
obj#=206418 tim=1422207498049
WAIT #39: nam='db file scattered read' ela= 1075 file#=147 block#=897442 blocks=7
obj#=206418 tim=1422207499416
WAIT #39: nam='db file scattered read' ela= 1649 file#=147 block#=897449 blocks=3
obj#=206418 tim=1422207501237
WAIT #39: nam='db file scattered read' ela= 2768 file#=147 block#=897453 blocks=4
obj#=206418 tim=1422207504191
WAIT #39: nam='db file scattered read' ela= 653 file#=147 block#=897458 blocks=7
obj#=206418 tim=1422207505141
WAIT #39: nam='db file scattered read' ela= 1588 file#=147 block#=897465 blocks=8
obj#=206418 tim=1422207507029
WAIT #39: nam='db file scattered read' ela= 460 file#=147 block#=897474 blocks=7
obj#=206418 tim=1422207507787
```

```
WAIT #39: nam='db file scattered read' ela= 608 file#=147 block#=897481 blocks=8
obj#=206418 tim=1422207508697
WAIT #39: nam='db file scattered read' ela= 564 file#=147 block#=897490 blocks=7
obj#=206418 tim=1422207509571
WAIT #39: nam='db file scattered read' ela= 832 file#=147 block#=897497 blocks=8
obj#=206418 tim=1422207510668
WAIT #39: nam='db file scattered read' ela= 846 file#=148 block#=102411 blocks=16
obj#=206418 tim=1422207512030
WAIT #39: nam='db file scattered read' ela= 4872 file#=148 block#=102427 blocks=16
obj#=206418 tim=1422207517488
WAIT #39: nam='db file scattered read' ela= 1624 file#=148 block#=102443 blocks=16
obj#=206418 tim=1422207520062
```

确实存在大量的 db file scattered read。这更加坚信了我的观点，表里存在大量的碎片。找第一个 db file scattered read的参数 `file#=146 block#=92900`，让客户执行 `ALTER system dump datafile 146 block min 92900 block max 92904`，获得的结果如下：

```
data_block_dump,data header at 0x6000000000208e64
===============
tsiz: 0x1f98
hsiz: 0x4c
pbl: 0x6000000000208e64
bdba: 0x24816ae4
     76543210
flag=--------
ntab=1
nrow=29
frre=0
fsbo=0x4c
fseo=0xf7
avsp=0x1f4c
tosp=0x1f4c
0xe:pti[0]      nrow=29 offs=0
0x12:pri[0]     sfll=1
0x14:pri[1]     sfll=2
0x16:pri[2]     sfll=3
0x18:pri[3]     sfll=4
0x1a:pri[4]     sfll=5
0x1c:pri[5]     sfll=6
0x1e:pri[6]     sfll=7
0x20:pri[7]     sfll=8
0x22:pri[8]     sfll=9
0x24:pri[9]     sfll=10
0x26:pri[10]    sfll=11
0x28:pri[11]    sfll=12
0x2a:pri[12]    sfll=13
0x2c:pri[13]    sfll=14
0x2e:pri[14]    sfll=15
0x30:pri[15]    sfll=16
0x32:pri[16]    sfll=17
0x34:pri[17]    sfll=18
0x36:pri[18]    sfll=19
0x38:pri[19]    sfll=20
```

```
0x3a:pri[20]    sfll=21
0x3c:pri[21]    sfll=22
0x3e:pri[22]    sfll=23
0x40:pri[23]    sfll=24
0x42:pri[24]    sfll=25
0x44:pri[25]    sfll=26
0x46:pri[26]    sfll=27
0x48:pri[27]    sfll=28
0x4a:pri[28]    sfll=-1
block_row_dump:
end_of_block_dump
```

里面全部是空记录。建议客户做一个 ALTER TABLE <table> MOVE;表重组后，发现原来 12GB 的表只剩下 800MB 了。再执行这个 SQL，只有 12 个缓冲区获取了：

```
Statistics
----------------------------------------------------------
          1  recursive calls
          0  db block gets
         12  consistent gets
          1  physical reads
          0  redo size
      18921  bytes sent via SQL*Net to client
        558  bytes received via SQL*Net from client
          8  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
         99  rows processed
```

解决问题后，我又和客户聊了几句，主要是关于碎片产生的原因的。我觉得如果真的按照客户所说，是不可能存在碎片的，肯定是这张表做了大量删除操作。最后，客户终于承认，这张表刚刚被导入，没有做任何删除操作也是听他手下的一个程序员说的，他也被错误的信息蒙蔽了。

在网上帮客户解决了几个小问题，不知不觉过去了 3 个多小时。在贵宾室吃过午饭后，广播里发出了登机通知。沈阳，我又回来了，只是不知道下飞机后要面对的又将是什么。

今日点评

今天老白处理这个案例时没有走太多弯路，主要原因是老白对相关理论有足够的了解。客户实际上提供了一个错误的信息，如果这时候你对表的内部存储原理、表碎片产生的原因、全表扫描的基本原理等理论知识不了解，那么这个问题可能会困扰你半天。

在分析问题时，如果你对主要的理论掌握得十分充分，那么除了根据经验分析问题外，还可以通过理论去进行分析。以这个案例为例，首先这个 SQL 是对全表进行扫描，因此会从表的第一块数据开始扫描，直到找到前 99 条记录为止。这个数据库的一个块大小为 8KB，一个块可以存放的数据大概会有几十条，那么 100 条记录只需要扫描几个数据块就可以完成了。如果 SQL*Plus 的默认数组大小是 5，那么顶多二三十个缓冲区获取就应该能够完成，而实际上产生了 68 538 个缓冲区获取。这从理论上只有一个解释，就是这张表的前面部分都是碎片，也就是格式化过的，

不过没有任何数据的块。所以老白要求客户马上转储几个数据块出来看看。通过转储的结果，很容易发现这些数据块都是空的。

优化小技巧 如何转储数据块

通过转储数据块来分析数据结构是 DBA 经常会用到的手段。今天的例子就是通过转储数据块发现表碎片的。现在通过一个实际的案例介绍一下如何转储数据块，并且简单地介绍一下数据块的内部结构。

这里介绍一下 `type: 0x06=trans data` 数据块的结构（本文涉及的数据库版本为 Oracle 10.2.0.1)。下面是一个块的逻辑结构：

```
buffer tsn: 4 rdba: 0x0100a714 (4/42772)
scn: 0x0000.0025486e seq: 0x02 flg: 0x04 tail: 0x486e0602
frmt: 0x02 chkval: 0xddf1 type: 0x06=trans data
Block header dump: 0x0100a714
Object id on Block? Y
seg/obj: 0xcc5c csc: 0x00.25486d itc: 3 flg: E typ: 1 - DATA
brn: 0 bdba: 0x100a711 ver: 0x01 opc: 0
inc: 0 exflg: 0

Itl Xid Uba Flag Lck Scn/Fsc
0x01 0xffff.000.00000000 0x00000000.0000.00 C--- 0 scn 0x0000.0025486d
0x02 0x0000.000.00000000 0x00000000.0000.00 ---- 0 fsc 0x0000.00000000
0x03 0x0000.000.00000000 0x00000000.0000.00 ---- 0 fsc 0x0000.00000000

data_block_dump,data header at 0x8eb227c
===============
tsiz: 0x1f80
hsiz: 0x16
pbl: 0x08eb227c
bdba: 0x0100a714
76543210
flag=----ntab=1 ----有 1 个表，如果是 CLUSTER，可能大于 1
nrow=2 ----有 2 行
frre=-1
fsbo=0x16 ----空闲空间的开始（44+NO_OF_ITLS*24+0x16），其中 NO_OF_ITLS 表示 itl 插槽的数量
fseo=0x1e70 ----空闲空间的尾部（44+NO_OF_ITLS*24 +0x1e70）
avsp=0x1e5a
tosp=0x1e5a
0xe:pti[0] nrow=2 offs=0
0x12:pri[0] offs=0x1f79 ----第一行的地址为 0x1f79+(44+NO_OF_ITLS*24)
0x14:pri[1] offs=0x1e70 ----第二行的地址为 0x1e70+(44+NO_OF_ITLS*24)
block_row_dump:
tab 0, row 0, @0x1f79 ----第一行
tl: 7 fb: --H-FL-- lb: 0x0 cc: 2
col 0: *NULL*
col 1: [ 2] c1 02 ----数值 1
tab 0, row 1, @0x1e70 ----第二行
tl: 263 fb: --H-FL-- lb: 0x0 cc: 2
col 0: [254]
```

```
31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35
36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 39 30
31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35
36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 39 30
31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35
36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 39 30
31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35
36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 39 30
31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35
36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 39 30
31 32 33 34
col 1: [ 2] c1 03 --数值2
end_of_block_dump
```

下面我们通过裸文件来分析。使用 dd 命令可以将这个数据块的信息转储出来（如果在 Windows 下需要使用 dd 工具，可以安装 Cygwin。它是一个在 Windows 下模拟 Linux Shell 的软件，安装后可以执行大多数 Linux 下的命令）。

首先我们看看前面的 20 个字节，这 20 个字节是：

```
06 A2 00 00 14 A7 00 01 6E 48 25 00 00 00 02 04
F1 DD 00 00
```

对应的数据结构为 kcbh（前面已经介绍过这个结构，这里不再重复）。

其中第 0 个字节为类型，06 表示是数据块。第 1 个字节是 FRMT，一般是 0x02，这里是 0xA2。用掩码 0x0f 取出就是 0x02。第 4～7 个字节为 14 A7 00 01，是 rdba 0x100A714(4/42772)，就是这个块。第 8～13 个字节是 SCN（前面 4 个是 BASE,后面是 WRAP）：6E 48 25 00 00 00 即十六进制的 0x254864.00。第 14 个字节 02 是 SEQ：SEQ=0x02。第 15 个字节 04 是标志：FLAG=0x04。第 16～17 个字节 F1 DD 是校验和 0xddf1。

从第 20 个字节开始的 24 字节是

```
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
F1 DD 00 00 01 00 00 00 5C CC 00 00 6D 48 25 00
00 00 00 00 03 00 32 00 11 A7 00 01 FF FF 00 00
```

其中第 20 个字节 01 表示 type=01。第 24～27 个字节 5C CC 00 00 表示十六进制的 0xcc5c（注意，这里有 UB4 的字节对齐问题，所以前面空了 3 个字节）。第 28～35 个字节是 6D 48 25 00 00 00 00 00，表示 SCN 为 0x0025486d.0000。第 36～37 个字节表示 ITL 插槽的数量，用 0x00ff 掩码。本例中为 03 00，表示有 3 个 ITL 槽。第 38 个字节是标志：FLAG=32。第 39 个字节 00 表示 FSL。第 40～43 个字节 11 A7 00 01 表示下一个 free 块的地址，即 rdba 0x100a711。

从第 44 个字节开始是 ITL 表，每个 ITL 记录的长度为 24。从前面可以看到这个块有 3 个 ITL 槽，因此 ITL 占 72 个字节。

下面我们开始分析：

```
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00 00 00 00 03 00 32 00 11 A7 00 01 FF FF 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 80 00 00
6D 48 25 00 00 00 00 00 00 00 00 00 00 00 00 00
```

其中第 44～51 个字节为 XID（8 个字节）：FF FF 00 00 00 00 00 00，表示 0xffff.0000.00000000。第 52～59 个字节为 UBA（8 个字节）：00 00 00 00 00 00 00 00，即 0x00000000.0000.00。第 60～61 个字节(2 个字节)：00 80 为标志，表示十六进制的 0x0800。第 62～67 个字节为 scn:00 00 6D 48 25 00，即 0x00254864.00。这个块有两个 ITL 槽，今后计算所有位置都要根据这个来计算。

下面我们来看这部分数据：

```
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00 00 00 00 00 01 01 00 FF FF 14 00 8E 1F 7A 1F
7a 1f 00 00 01 00 8e 1f
```

其中第 100 个字节表示 FLAG=0(--------------)。第 101 个字节表示 NTAB 为 1，块中有一张表的数据。第 102～103 个字节 01 00 表示 nrow=1，块中有一条记录。第 104～105 个字节为 ff ff。第 106～107 个字节为 14 00，即十六进制的 0x14，块中空闲空间的起始。第 108～109 个字节为 8e 1f，即十六进制的 0x1f8e，块中空闲空间的终止。第 110～111 个字节为 7a 1f，即十六进制的 0x1f7a，块中可用空间。第 112～113 个字节为 7a 1f，即十六进制的 0x1f7a。

优化小技巧　通过树转储来分析索引碎片

在前面的优化方案设计时，我们谈到了重建索引的问题。由于索引是 B 树结构，所以经过一定时间的数据变更，会由于节点分裂而导致出现较多的碎片。我在前面说过，通过查看索引的大小来判断是否存在碎片。不过这种方法比较粗略，如果要详细了解索引碎片的情况，可以通过树转储来进行分析。下面就通过树转储来分析索引分裂。

下面分析 B 树索引（本次分析是在自动段存储管理模式下进行的）。

1. 创建测试表

首先创建一张测试表，然后使用 dba-object 中的数据创建一些测试数据：

```
SQL> CREATE sequence seq1 ;
序列已创建。

SQL> CREATE table tobj AS SELECT seq1.nextval seq1,object_id,object_name FROM
dba_objects WHERE rownum<8000;
表已创建。

SQL>  CREATE index idx_tobj on tobj(seq1) pctfree 0;
索引已创建。
```

2. 转储索引的 B 树

首先找到这个索引对应的 OBJECT_ID：

```
SQL> SELECT object_id,object_name FROM user_objects WHERE object_name LIKE 'IDX_%';
 OBJECT_ID OBJECT_NAME
---------- ----------------------------------------
     32147 IDX_TTA2
     32019 IDX_T_SAMPLE1
```

然后通过 treedump 事件将索引的树结构转储出来：

```
SQL> ALTER session set events 'immediate trace name treedump level 32147';
会话已更改。
```

查看 Trace 文件：

```
----- begin tree dump
branch: 0x14039f4 20986356 (0: nrow: 31, level: 1)
   leaf: 0x14039f5 20986357 (-1: nrow: 267 rrow: 267)
   leaf: 0x14039f6 20986358 (0: nrow: 260 rrow: 260)
   leaf: 0x14039f7 20986359 (1: nrow: 260 rrow: 260)
   leaf: 0x14039f8 20986360 (2: nrow: 260 rrow: 260)
   leaf: 0x14039f9 20986361 (3: nrow: 260 rrow: 260)
   leaf: 0x14039fa 20986362 (4: nrow: 260 rrow: 260)
   leaf: 0x14039fb 20986363 (5: nrow: 260 rrow: 260)
   leaf: 0x14039fc 20986364 (6: nrow: 260 rrow: 260)
   leaf: 0x14039fd 20986365 (7: nrow: 260 rrow: 260)
   leaf: 0x14039fe 20986366 (8: nrow: 260 rrow: 260)
   leaf: 0x14039ff 20986367 (9: nrow: 260 rrow: 260)
   leaf: 0x1403a00 20986368 (10: nrow: 260 rrow: 260)
   leaf: 0x1403a02 20986370 (11: nrow: 260 rrow: 260)
   leaf: 0x1403a03 20986371 (12: nrow: 260 rrow: 260)
   leaf: 0x1403a04 20986372 (13: nrow: 260 rrow: 260)
   leaf: 0x1403a05 20986373 (14: nrow: 260 rrow: 260)
   leaf: 0x1403a06 20986374 (15: nrow: 260 rrow: 260)
   leaf: 0x1403a07 20986375 (16: nrow: 260 rrow: 260)
   leaf: 0x1403a08 20986376 (17: nrow: 260 rrow: 260)
   leaf: 0x1403a09 20986377 (18: nrow: 260 rrow: 260)
   leaf: 0x1403a0a 20986378 (19: nrow: 260 rrow: 260)
   leaf: 0x1403a0b 20986379 (20: nrow: 260 rrow: 260)
   leaf: 0x1403a0c 20986380 (21: nrow: 260 rrow: 260)
   leaf: 0x1403a0d 20986381 (22: nrow: 260 rrow: 260)
   leaf: 0x1403a0e 20986382 (23: nrow: 260 rrow: 260)
   leaf: 0x1403a0f 20986383 (24: nrow: 260 rrow: 260)
   leaf: 0x1403a10 20986384 (25: nrow: 260 rrow: 260)
   leaf: 0x1403a12 20986386 (26: nrow: 260 rrow: 260)
   leaf: 0x1403a13 20986387 (27: nrow: 260 rrow: 260)
   leaf: 0x1403a14 20986388 (28: nrow: 260 rrow: 260)
   leaf: 0x1403a15 20986389 (29: nrow: 192 rrow: 192)
----- end tree dump
```

下面看看分支节点：

```
branch: 0x14039f4 20986356 (0: nrow: 31, level: 1)   (5/14836)
```

我们找到分支节点的 RDBA 为(5/14836)。下一步通过块转储，看看这个分支节点的逻辑数据：

```
SQL> alter system dump datafile 5 block 14836;
系统已更改。
```

下面是转储出来的数据块逻辑结构：

```
kdxbrlmc 20986357=0x14039f5
kdxbrsno 0
kdxbrbksz 3964
```

```
row#0[3955] dba: 20986358=0x14039f6
col 0; len 3; (3):  c2 03 45
col 1; TERM
row#1[3946] dba: 20986359=0x14039f7
col 0; len 3; (3):  c2 06 1d
col 1; TERM
row#2[3937] dba: 20986360=0x14039f8
col 0; len 3; (3):  c2 08 59
col 1; TERM
row#3[3928] dba: 20986361=0x14039f9
col 0; len 3; (3):  c2 0b 31
col 1; TERM
row#4[3919] dba: 20986362=0x14039fa
col 0; len 3; (3):  c2 0e 09
col 1; TERM
row#5[3910] dba: 20986363=0x14039fb
col 0; len 3; (3):  c2 10 45
col 1; TERM
row#6[3901] dba: 20986364=0x14039fc
col 0; len 3; (3):  c2 13 1d
col 1; TERM
row#7[3892] dba: 20986365=0x14039fd
col 0; len 3; (3):  c2 15 59
col 1; TERM
row#8[3883] dba: 20986366=0x14039fe
col 0; len 3; (3):  c2 18 31
col 1; TERM
row#9[3874] dba: 20986367=0x14039ff
col 0; len 3; (3):  c2 1b 09
col 1; TERM
row#10[3865] dba: 20986368=0x1403a00
col 0; len 3; (3):  c2 1d 45
col 1; TERM
row#11[3856] dba: 20986370=0x1403a02
col 0; len 3; (3):  c2 20 1d
col 1; TERM
row#12[3847] dba: 20986371=0x1403a03
col 0; len 3; (3):  c2 22 59
col 1; TERM
row#13[3838] dba: 20986372=0x1403a04
col 0; len 3; (3):  c2 25 31
col 1; TERM
row#14[3829] dba: 20986373=0x1403a05
col 0; len 3; (3):  c2 28 09
col 1; TERM
row#15[3820] dba: 20986374=0x1403a06
col 0; len 3; (3):  c2 2a 45
col 1; TERM
row#16[3811] dba: 20986375=0x1403a07
col 0; len 3; (3):  c2 2d 1d
col 1; TERM
row#17[3802] dba: 20986376=0x1403a08
col 0; len 3; (3):  c2 2f 59
col 1; TERM
```

```
row#18[3793] dba: 20986377=0x1403a09
col 0; len 3; (3):  c2 32 31
col 1; TERM
row#19[3784] dba: 20986378=0x1403a0a
col 0; len 3; (3):  c2 35 09
col 1; TERM
row#20[3775] dba: 20986379=0x1403a0b
col 0; len 3; (3):  c2 37 45
col 1; TERM
row#21[3766] dba: 20986380=0x1403a0c
col 0; len 3; (3):  c2 3a 1d
col 1; TERM
row#22[3757] dba: 20986381=0x1403a0d
col 0; len 3; (3):  c2 3c 59
col 1; TERM
row#23[3748] dba: 20986382=0x1403a0e
col 0; len 3; (3):  c2 3f 31
col 1; TERM
row#24[3739] dba: 20986383=0x1403a0f
col 0; len 3; (3):  c2 42 09
col 1; TERM
row#25[3730] dba: 20986384=0x1403a10
col 0; len 3; (3):  c2 44 45
col 1; TERM
row#26[3721] dba: 20986386=0x1403a12
col 0; len 3; (3):  c2 47 1d
col 1; TERM
row#27[3712] dba: 20986387=0x1403a13
col 0; len 3; (3):  c2 49 59
col 1; TERM
row#28[3703] dba: 20986388=0x1403a14
col 0; len 3; (3):  c2 4c 31
col 1; TERM
row#29[3694] dba: 20986389=0x1403a15
col 0; len 3; (3):  c2 4f 09
col 1; TERM
----- end of branch block dump -----
End dump data blocks tsn: 5 file#: 5 minblk 14836 maxblk 14836
```

这个枝节点中共有 30 行，索引到 31 个叶节点。这个结果和 treedump 的结果类似。

我们来看第 0 行：

```
row#0[3955] dba: 20986358=0x14039f6
col 0; len 3; (3):  c2 03 45
col 1; TERM
```

对应的值是 `c2 03 45`（对应的十进制为 268），对应的 DBA 是 `0x14039f6`。

3. 普通叶节点分裂分析

首先通过插入数据来使索引数据发生一些变化：

```
SQL> insert into tobj values (260,123,'sss');
已创建 1 行。
```

```
SQL> commit;
提交完成。
```

然后再次转储 B 树：

```
SQL> ALTER SESSION SET EVENTS 'immediate trace name treedump level 32147';
会话已更改。
```

我们看看数据发生了什么样的变化：

```
----- begin tree dump
branch: 0x14039f4 20986356 (0: nrow: 32, level: 1)
   leaf: 0x14039f5 20986357 (-1: nrow: 142 rrow: 142)          叶节点分裂
   leaf: 0x1403a19 20986393 (0: nrow: 126 rrow: 126)
   leaf: 0x14039f6 20986358 (1: nrow: 260 rrow: 260)
   leaf: 0x14039f7 20986359 (2: nrow: 260 rrow: 260)
   leaf: 0x14039f8 20986360 (3: nrow: 260 rrow: 260)
   leaf: 0x14039f9 20986361 (4: nrow: 260 rrow: 260)
   leaf: 0x14039fa 20986362 (5: nrow: 260 rrow: 260)
   leaf: 0x14039fb 20986363 (6: nrow: 260 rrow: 260)
   leaf: 0x14039fc 20986364 (7: nrow: 260 rrow: 260)
   leaf: 0x14039fd 20986365 (8: nrow: 260 rrow: 260)
   leaf: 0x14039fe 20986366 (9: nrow: 260 rrow: 260)
   leaf: 0x14039ff 20986367 (10: nrow: 260 rrow: 260)
   leaf: 0x1403a00 20986368 (11: nrow: 260 rrow: 260)
......
   leaf: 0x1403a14 20986388 (29: nrow: 260 rrow: 260)
   leaf: 0x1403a15 20986389 (30: nrow: 192 rrow: 192)
----- end tree dump
```

从前面这几行数据：

```
----- begin tree dump
branch: 0x14039f4 20986356 (0: nrow: 32, level: 1)
   leaf: 0x14039f5 20986357 (-1: nrow: 142 rrow: 142)
   leaf: 0x1403a19 20986393 (0: nrow: 126 rrow: 126)
```

如果我们把这些数据和插入数据前的转储数据相比，就可以看出此时产生了叶节点分裂。

4. 看看删除的记录是否能被重用

首先我们重新准备测试数据，删除刚才的测试索引，并且重建索引。为了能够很快看到叶节点分裂的情况，创建索引时，我们将 pctfree 参数设置为 0：

```
DROP index idx_tobj;
CREATE index idx_tobj on tobj(seq1) pctfree 0;
DELETE tobj WHERE seq1=260;
INSERT INTO tobj VALUES (260,260,'sss');
SELECT object_id,object_name FROM user_objects WHERE object_name LIKE 'IDX%';

SQL> ALTER SESSION SET EVENTS 'immediate trace name treedump level 32148';
会话已更改。
```

此时我们得到的索引的树结构如下：

```
branch: 0x14039f4 20986356 (0: nrow: 32, level: 1)
   leaf: 0x14039f5 20986357 (-1: nrow: 142 rrow: 142)
   leaf: 0x1403a19 20986393 (0: nrow: 126 rrow: 124)
   leaf: 0x14039f6 20986358 (1: nrow: 260 rrow: 260)
   leaf: 0x14039f7 20986359 (2: nrow: 260 rrow: 260)
   leaf: 0x14039f8 20986360 (3: nrow: 260 rrow: 260)
   leaf: 0x1403a14 20986388 (29: nrow: 260 rrow: 260)
   leaf: 0x1403a15 20986389 (30: nrow: 193 rrow: 193)
----- end tree dump
```

看来叶节点分裂还是发生了。说明删除记录的空间无法被使用，这个和 Oracle 概念里是相同的。下面我们测试一下，看看删除某个块中的绝大多数值是什么样的：

```
----- begin tree dump
branch: 0x14039f4 20986356 (0: nrow: 31, level: 1)
   leaf: 0x14039f5 20986357 (-1: nrow: 267 rrow: 267)
   leaf: 0x14039f6 20986358 (0: nrow: 260 rrow: 260)
```

首先转储出叶节点 0x14039f5（5，14837）的逻辑数据：

```
row#0[3928] flag: -----, lock: 0
col 0; len 2; (2):  c1 02                        /*第一行，十进制 1*/
col 1; len 6; (6):  01 40 39 94 00 00


row#266[570] flag: -----, lock: 0
col 0; len 3; (3):  c2 03 44                     /*最后一行，十进制 267*/
col 1; len 6; (6):  01 40 39 97 00 0d
```

然后我们删除绝大多数数据：

```
DELETE tobj WHERE seq1<267;
```

下面来看这个叶节点的信息：除了最后一行外，其他的都是标志---D-，说明是被删除的行。

```
row#0[3928] flag: ---D-, lock: 2
col 0; len 2; (2):  c1 02
col 1; len 6; (6):  01 40 39 94 00 00
row#1[3916] flag: ---D-, lock: 2
col 0; len 2; (2):  c1 03
col 1; len 6; (6):  01 40 39 94 00 01
row#2[3904] flag: ---D-, lock: 2
col 0; len 2; (2):  c1 04
col 1; len 6; (6):  01 40 39 94 00 02
row#3[3892] flag: ---D-, lock: 2
col 0; len 2; (2):  c1 05
col 1; len 6; (6):  01 40 39 94 00 03
row#4[3880] flag: ---D-, lock: 2
col 0; len 2; (2):  c1 06
col 1; len 6; (6):  01 40 39 94 00 04
row#5[3868] flag: ---D-, lock: 2
col 0; len 2; (2):  c1 07
```

```
col 1; len 6; (6):  01 40 39 94 00 05
row#6[3856] flag: ---D-, lock: 2
col 0; len 2; (2):  c1 08
col 1; len 6; (6):  01 40 39 94 00 06
......
row#264[596] flag: ---D-, lock: 2
col 0; len 3; (3):  c2 03 42
col 1; len 6; (6):  01 40 39 97 00 0b
row#265[583] flag: ---D-, lock: 2
col 0; len 3; (3):  c2 03 43
col 1; len 6; (6):  01 40 39 97 00 0c
row#266[570] flag: -----, lock: 0
col 0; len 3; (3):  c2 03 44
col 1; len 6; (6):  01 40 39 97 00 0d
----- end of leaf block dump -----
End dump data blocks tsn: 5 file#: 5 minblk 14837 maxblk 14837
```

这个时候再插入一行：

```
INSERT INTO tobj VALUES(100,1,'sss');
commit;
```

然后再看看 treedump 的结果：

```
branch: 0x14039f4 20986356 (0: nrow: 31, level: 1)
   leaf: 0x14039f5 20986357 (-1: nrow: 2 rrow: 2)
   leaf: 0x14039f6 20986358 (0: nrow: 260 rrow: 260)
   leaf: 0x14039f7 20986359 (1: nrow: 260 rrow: 260)
   leaf: 0x14039f8 20986360 (2: nrow: 260 rrow: 260)
```

这时并没有发生叶节点分裂。看看页块中的信息：

```
row#0[558] flag: -----, lock: 2
col 0; len 2; (2):  c2 02
col 1; len 6; (6):  01 40 39 95 00 55
row#1[570] flag: -----, lock: 0
col 0; len 3; (3):  c2 03 44
col 1; len 6; (6):  01 40 39 97 00 0d
----- end of leaf block dump -----
End dump data blocks tsn: 5 file#: 5 minblk 14837 maxblk 14837
```

叶节点被重新格式化了。这好像和概念中的不同，我们再来分析一下什么时候会产生重新格式化呢？

5. 看看叶节点重新格式化的条件

首先我们重新创建测试环境，重建表和序列号：

```
SQL> drop sequence seq1;
序列已丢弃。

SQL> create sequence seq1;
序列已创建。
```

```
SQL> drop table tobj;
表已丢弃。

SQL> create table tobj  as select seq1.nextval seq1,object_id,object_name from
dba_objects where rownum<8000;
表已创建。

SQL> create index idx_tobj on tobj(seq1) pctfree 0;
索引已创建。

SQL> SELECT object_id,object_name FROM user_objects WHERE object_name LIKE 'IDX%';

 OBJECT_ID OBJECT_NAME
---------- ----------------------------------------
     32183 IDX_TOBJ
     32100 IDX_TTA2
     32019 IDX_T_SAMPLE1
```

这时转储出索引的树结构，应该和前面实验看到的一样。下面我们的实验该变了，删除记录后再提交，然后再次插入数据，看看结果会如何呢？

```
SQL> DELETE tobj WHERE Seq1=100;
已删除 1 行。

SQL> commit;
提交完成。

SQL> INSERT INTO tobj VALUES (100,1,'sss');
已创建 1 行。

SQL> commit;
提交完成。
```

我们来看看这次实验的 treedump：

```
branch: 0x14039f4 20986356 (0: nrow: 31, level: 1)
   leaf: 0x14039f5 20986357 (-1: nrow: 267 rrow: 267)
   leaf: 0x14039f6 20986358 (0: nrow: 260 rrow: 260)
   leaf: 0x14039f7 20986359 (1: nrow: 260 rrow: 260)
   leaf: 0x14039f8 20986360 (2: nrow: 260 rrow: 260)
   leaf: 0x14039f9 20986361 (3: nrow: 260 rrow: 260)
```

从 treedump 中，我们并没有看到叶节点分裂，那么再来看看第一个叶节点的信息：

```
row#99[2740] flag: -----, lock: 0
col 0; len 2; (2):  c2 02
col 1; len 6; (6):  01 40 39 95 00 0d
```

这个槽位被重用了。从这可以看出，一旦提交，索引立即被重组。被删除的记录所占用的空间立即可以被重用。但是，如果事务中没有提交前插入的行不能使用刚才被删除行留下来的空闲位置，那么在数据块已满的情况下，就会导致叶节点分裂。

为什么 Oracle 不马上把删除的数据留下的空闲空间利用起来呢？实际上这个问题很好理解。

如果这么做，那么一旦这个事务回滚了，回滚操作的成本是不是会变得十分高呢？

实际上，从上面的实验我们可以学到一些优化技巧。如果一张表经常会进行删除、插入操作，那么这张表上相关的索引可能经常会出现叶节点分裂，严重时索引的碎片化会很严重，影响该索引的性能。对于这种情况，适当减少批量提交，可以减轻索引碎片化。如果我们的应用无法调整，那么经常对这类索引进行重建或者在线重建，也是十分必要的。

6月17日 完美的效果

昨天由于飞机晚点，回到沈阳已经比较晚了，所以就直接回宾馆休息了。晚上和老于碰了一下，有一个十分好的消息，就是那个该死的12张表连接的SQL终于修改完毕了。开发商在测试环境测试了，效果十分好。去掉占位操作后，执行计划十分完美，开销减少到了原来的万分之一以下。

昨天说好了，开发商计划16日晚上正式发布这个新补丁，因此今天早上一上班，我和老于迫不及待地连到账务系统上，查看系统的运行情况。账务系统的CPU资源一直存在较为严重的瓶颈，不过从今天早上的现象来看，CPU资源得到了很大的缓解，平均使用率从以前的超过90%下降为60%左右。通过STATSPACK报告查看那个12张表连接的SQL，发现平均每次执行产生的缓冲区获取为64，物理读为2，执行时间为33s。而在优化前，这个SQL的开销是平均为此执行140万逻辑读，55万物理读，执行时间为639s。

老于打了个电话给开发商，向他们表示祝贺，开发商那边也十分高兴。昨天三套系统中优化完成的SQL都已经发布，从BILL、CAL和BSS三套系统今天上午的表现来看，效果十分好。如果这种现象能够保持，那么这个项目应该可以圆满结束了。今天所有的优化工作都能够测试完毕，晚上将发布剩下的十多个模块。如果明天上午能够确认优化效果，开发商建议向甲方提出验收申请。

我坐到小孙身旁，让他打开BBS，看看上面是否有一些好的反馈。这几天BBS上人气不旺，只有少量的信息，不过今天早上有两个帖子，说发现今天系统很快，帖子的标题上还有一个很明显的笑脸。我让小孙以优化小组的名义发了一个帖子，说优化工作阶段性完成，希望大家反馈对系统性能的评价。帖子发出不到半小时，就有上百个人跟帖。从跟帖上来看，大多数都是正面的，甚至还有人喊出了“万岁”的口号。不过也有不少人提出，以前也做过优化，不过效果总是不能持久，不知道这次是回光返照还是确实系统调好了。

我打电话给张工，希望他能够找几个营业厅，打电话过去询问一下。张工还不知道昨天晚上上线了大部分模块，还有点不放心，希望我能够确认一下，怕电话打过去不是那么回事，还被营业厅恶心一把。我让他放心，优化效果是通过多方面确认过的，绝对不会出问题。张工找了几个业务量比较大的营业厅，通过电话进行了确认。营业厅那边的反馈都很不错，连平时总是排长队的几个营业厅都感觉今天的业务处理速度快了很多，看不到往常的长队了。我说再问问异地的营业厅吧。

过了 10 分钟，张工突然打电话来问，是不是现在系统又出问题了，营口那边反馈系统突然变慢了，而且比以前还慢。我和老于急忙跑到办公室。从 OEM 的监控上看，数据库很正常，操作系统的各项指标也和刚上班时一样。

我立即联系了营口那边的维护人员，从他那边反馈的信息是早上一上班系统很不错，速度比平时快了不少，不过 1 个小时前，系统突然变慢了，而且一下子变得特别慢。他也测试过，通过 SQL*Plus 连接数据库也很慢，而且连上去后做 SELECT 操作也十分慢。我建议他马上连上去，做一个 10 级的 10046 Trace，看看主要的等待事件是什么。在电话中经过 10 分钟的沟通，终于将 ALTER SYSTEM SET EVENTS='10046 TRACE NAME CONTEXT FOREVER,LEVEL 10'的命令执行完毕。从 Trace 文件上看，大量的等待事件是 SQL*Net 系列的，比如 SQL*Net more data to client 等。我这边还在和营口的维护人员打电话，老于已经打开了 Trace 文件看了起来，说："老白，让他们查查网络，都是 SQL*Net 的等待。"我马上让他们做一个 ping 服务器的操作，普通的 ping 很正常，不过 ping 大包时，包丢失情况很严重，大概有 50%以上的包会丢失。该不是中了冲击波病毒了吧，看着这个情况，我的第一反应是病毒。

半个小时后，营口那边确认了，有几台微机染上了冲击波病毒。关闭那几台机器后，系统恢复了正常。被营口的事情折腾了小半天，安静下来后才发现，已经到中午了。吃完午饭，老于和小齐联系了一下，把这边的情况简单说了说。小齐听了半天，只问了一句话："我是不是可以向集团公司申请验收了？"老于很坚决地说："可以了，看样子我们的优化工作可以告一段落了，准备 7 月上旬验收吧。"

下午系统比较平稳，我和老于基本上没进办公室，每个人抱着一杯茶，坐在吸烟区抽烟聊天。经过 2 个月的紧张工作，我和老于都很需要放松放松。下午孙主任打电话给老于，询问了目前系统的状态，并且通知我们明天下午到总部开会，讨论内容是下一步的工作安排。

老于正在和孙主任通电话，小齐的电话就打进来了。原来小齐已经把情况汇报给集团公司，集团公司建议尽快安排下一步的工作计划，因为辽宁只是集团公司的优化试点，根据辽宁的情况，将确定下一步北方公司其他省份的工作安排。由于这个项目必须在春节前全部完成，因此集团公司领导比我们还急。明天的会议，集团公司的相关领导将以电话会议的形式参与。我和小齐说这个会能不能推迟一天开，因为今天晚上还有部分模块上线，如果明天的效果很好，这个会开得比较踏实。小齐说集团公司领导后天一早就要出国考察，一周后才能回国，现在一点时间都耽误不起了。除非明天确实没有把握，否则这个会就必须明天开。我和老于必须马上确定明天这个会能不能开。

这时候老于也已经接完了孙主任的电话，和我同样茫然，估计他们讨论的问题与我和小齐的是相同的。

"开不开？"我问老于。这么大的事情，我也不敢轻易做决定。如果明天系统状态十分正常，那么这个会可以开得很好，不过如果系统真的出了问题，那么就很尴尬了。

"不开也得开了，这种情况没有任何理由说不开，希望应用不要出问题。"老于今天异常坚决，看样子确实是没有任何退路了。

确认了明天开会后，我和老于说我要去小王那边看看，对于今天晚上上线的模块，我要再确认一下，不要明天开会的时候，被投诉电话打断。

今天真是奇怪的一天，兴奋之余，更多的是担忧。已经忙了两个多月了，今天晚上才是最难熬的啊。

今日点评

今天优化的效果已经显现了，而老白他们又面临一个选择，就是明天的总结会开不开的问题。今天晚上要上线一些重要的模块，而明天的会议北方公司也要参加，如果真的在开会时发生一些问题，那么就很被动了。但是事实上客观条件决定这个会必须开。这种情况下，DBA 的判断就十分重要了，如果风险过大，那么明天的会议是坚决不能开的。正是因为老于和老白基于今天系统的情况，才做出了明天继续开会的决定。

不过为了确保明天这么重要的会议开得顺利，老白还是去开发小组那里做了一些确认工作。确认工作主要是对明天需要上线的模块涉及的一些主要 SQL 进行确认，包括对执行计划和开销的比对。确保明天这些 SQL 上线后不会对系统产生明显的副作用。

优化小技巧　I/O 系统优化的经验与体会

在海量数据的情况下，数据库的性能问题有 80%以上和 I/O 相关，因此，I/O 优化是贯穿海量数据库管理全过程的重要工作。I/O 优化牵涉的面比较广，现在就从 Oracle 数据库优化的一些主要方面详细阐述一下。在海量数据库环境下，Oracle 数据库优化面临的最重要的任务是 I/O 优化，因为绝大多数大型数据库都或多或少存在 I/O 性能问题。数据库的 I/O 性能牵涉面十分广，因此 I/O 性能优化也是 Oracle 数据库优化工作中最复杂的工作。进行 I/O 性能优化，基本的步骤是这样的：

- 采集系统数据，定位 I/O 性能问题；
- 分析并制订解决方案；
- 调整系统，解决问题；
- 评估优化结果。

优化 Oracle 数据库的 I/O 性能，不能简单地从 I/O 入手，需要遵循数据库优化工作的基本原则。首先数据库优化的目的是提高系统的整体处理能力，提高事务响应速度。所有的优化工作都需要围绕这个目的来进行。数据库性能优化的手段有两个方面，一是减少处理时间，二是减少等待时间。减少处理时间要求应用编写得合理高效。减少等待时间要求系统的各种资源合理分配，不出现任何瓶颈。数据库使用的系统资源包括 CPU、内存、存储和网络。数据库优化的目的是合理分配这些资源，确保任何一种资源都不出现瓶颈。

根据上述原则，Oracle 数据库的 I/O 性能优化不能只通过重新组合系统资源来达到提升数据库总体性能（包括 I/O 性能）的目的。另一个方面，在优化 I/O 时也要考虑到其他资源的情况。

如果 I/O 单方面提升导致其他资源出现瓶颈，那么也会导致系统总体性能下降。特别是针对资源有限的系统的优化，如何有效利用不足的系统资源进行最优化的组合，在保证没有一种资源过载的情况下尽可能利用资源，是系统优化的关键。

如何确定系统的主要问题是 I/O 问题，并进一步定位 I/O 问题的根本原因是解决 Oracle I/O 性能问题的关键。I/O 性能不佳，可能是多方面的问题。进行优化的第一步就是确定关键的性能瓶颈。

影响 Oracle 数据库 I/O 性能的问题覆盖面十分广，根据作者多年从事 Oracle 数据库管理的经验，下面几个方面是影响数据库 I/O 性能的主要问题。

- 存储性能瓶颈：控制器不足、Cache 偏小，Cache 设置不合理、I/O 通道容量不足等。
- 磁盘性能瓶颈：磁盘数量过少、使用了速度比较低的磁盘等。
- 使用了不合理的 RAID 模式。
- 在使用 RAID 的情况下，存在 I/O 热点，多个热点文件使用同一个磁盘。
- 异步 I/O 配置不正确。
- 数据库各种缓冲区设置不合理，缓冲命中率过低。
- PGA 的各种缓存设置过小（对于 Oracle 9i，在使用自动管理模式的情况下，PGA 设置过小），导致大量临时表空间操作。
- 重做日志存在性能瓶颈。
- 重做缓冲区设置不合理。
- 存在热点数据。
- 表空间碎片严重。
- 表和索引的存储参数不合理。
- 行迁移比较严重。
- 存在大量大表扫描的 SQL。
- SQL 执行选择了不好的执行计划。

当系统出现 I/O 问题时，数据库管理员最大的挑战是如何尽快找到问题的最根本的原因。I/O 问题的调整是十分复杂的，在没有找到根本原因前进行调整往往无法达到最终的优化目标。需要注意的是 I/O 问题往往和大型的 SQL 语句有关。如果某个系统突然发生 I/O 性能问题，第一步需要排除一切 I/O 以外的问题。

确定 I/O 性能瓶颈的存在，并定位存在 I/O 性能瓶颈的设备是解决 I/O 性能问题的第一步。使用操作系统的监控工具可以实时监控 I/O 的情况。第一种方法是使用 vmstat 工具。使用该工具，可以查看 b 列的值,如果该值比较大，那么说明等待 I/O 的进程比较多，I/O 可能存在问题，如下所示：

```
$vmstat 1 10
         procs           memory
    r     b      w      avm     free
    2    12      0  14572705   92752
```

```
2    12     0  14572705   93548
2    12     0  14572705   92910
2    12     0  14572705   93467
2    12     0  14572705   93546
2    12     0  14572705   93864
2    12     0  14572705   94557
2    12     0  14572705   93952
2    12     0  14572705   94017
2    12     0  14572705   93047
```

如果上述命令发现 b 比较大，那么说明可能存在 I/O 等待的现象，通过 sar 命令或者 iostat 命令可以进一步确认。如果用 sar 命令监控时发现 wio 比较大（比如超过 40，这个值是个经验值，根据不同的系统，这个值可以调整），那么基本上可以确定存在 I/O 性能瓶颈，如下所示：

```
$sar 1 10
HP-UX cuyn16 B.11.11 U 9000/800    11/01/05
15:01:44    %usr    %sys    %wio   %idle
15:01:45      16       3      57      24
15:01:46      15       2      59      24
15:01:47      21       3      57      19
15:01:48      20       2      63      16
15:01:49      17       2      67      14
15:01:50      12       1      75      12
15:01:51      16       2      75       7
15:01:52      10       1      84       5
15:01:53      18       2      79       1
15:01:54      25       3      65       6
```

如果发现 wio 值长时间高于 40，那么说明 I/O 等待比较严重。此时可以通过 sar-d 命令来监控，看看哪些 I/O 设备存在性能问题。如果发现某个设备的繁忙度长时间超过 90%，就说明该设备比较繁忙。如果该设备的 avserv 比较大或者比其他设备高很多，就说明该设备存在性能问题。比如下面的例子：

```
15:02:01    device   %busy   avque   r+w/s   blks/s  avwait  avserv
Average    c0t0d0    2.00    0.50       6      27    3.62    6.03
Average    c3t8d0    1.10    0.50       4      16    3.23    5.69
Average   c55t0d5   99.40    0.50      18      73    5.41   54.50
Average   c55t1d0    4.20    0.50       5      15    5.39    8.49
Average   c55t1d1   79.52    0.76      24     810    9.09   81.99
Average  c55t10d7   68.33    0.52      23    2909    5.60   72.40
Average  c55t11d0   31.07    1.14      25    1630   10.95   28.05
Average  c55t11d2   16.98    0.51      22    3075    5.24   13.39
Average  c55t11d3   71.83    2.59      26    1643   42.18   82.78
Average  c55t11d5   76.12    0.50      23    3012    5.58   76.47
Average  c55t11d6   30.57    1.02      26    1637   10.85   30.59
Average  c55t12d0   21.48    0.50      20    2826    5.12   19.55
Average  c55t12d1   80.72    2.74      29    1880   42.78   84.38
Average  c55t12d3   70.03    0.52      23    2887    5.83   66.85
Average  c55t12d4   23.58    1.74      27    1758   16.11   25.72
Average  c55t14d0   80.62    0.50      66     975    4.83   12.38
Average  c56t14d2    5.89    0.50       6      75    5.67   10.38
Average  c55t11d1   80.22    0.50      22    1547    4.85   75.33
```

```
Average  c55t11d4   24.08    0.50      26    1829    5.04   11.90
Average  c55t11d7   82.72    0.50      23    1586    5.27   76.46
Average  c55t12d2   24.18    0.50      18    1304    4.79   16.26
Average  c55t12d5   76.02    0.50      20    1403    4.91   74.11
Average  c55t14d1  100.00   54.57     104    6630 1315.98   71.54
Average  c55t13d1   77.72    0.55      19     297    5.79   80.19
Average   c55t1d4    0.30    0.50       1      10    7.02    1.50
Average   c55t2d0    0.10    0.50       0       3    3.59    6.43
Average   c55t6d2    0.30    0.50       0       6    5.17    7.09
Average   c55t0d7    0.70    0.50       0       2    1.33   69.67
Average  c55t12d6    0.10    0.50       0       0    0.98    9.41
Average  c55t13d0    0.50    0.50       0       4    7.46   19.99
```

从上面的数据可以看出，部分设备（比如 c55t14d1）的等待队列比较长，并且 avwait+avserv 的时间比较长，说明该设备存在性能瓶颈。而大部分 HDISK 的 avwait+avserv 还比较正常，这说明存储系统并不存在普遍性的性能瓶颈，而性能瓶颈主要集中在热点盘组上。

通过 Oracle 的 STATSPACK 工具也可以检查系统的 I/O 问题。如果系统的性能不佳，并且从报告中看到的 sequencial read 等待事件是系统等待事件中排在前几位的事件，占系统总等待时间的比例比较高，那么系统很可能存在 I/O 性能问题。可以通过检查文件 I/O 情况来进一步确认并找出存在性能问题的设备。方法是通过检查文件 I/O 中的平均读响应时间。如果某个文件的平均读响应时间超过 20ms，那么说明该文件所属的文件系统可能存在性能问题。应该注意的是，20ms 是一个相对参数，在不同的应用环境下该值可能会有所不同。通过比对操作系统的情况，数据库管理员应该很快就能确定你所管理的系统的平均读响应时间和操作系统 I/O 情况的对应关系。

检查读平均响应时间时还要注意几个问题。第一个问题是在报告时间区域内的 I/O 量，如果某个文件在报告时间区间内的 I/O 数量很小，那么此平均响应时间可能缺乏代表性，可以通过检查存放在相同设备上的其他文件来进一步确认。另外一种情况是平均每次读的数据量比较多（每次读的块数比较多），那么略高的平均读响应时间也是正常的。下面的数据就是从 I/O 存在问题的数据库上取下的 STATSPACK 数据：

```
Top 5 Timed Events
~~~~~~~~~~~~~~~~~~                                                     % Total
Event                                            Waits    Time (s) Ela Time
-------------------------------------------- ------------ ----------- --------
db file sequential read                           661,802      45,404    60.79
SQL*Net more data from dblink                   3,180,371       7,894    10.57
CPU time                                                        5,077     6.80
db file scattered read                             56,959       3,846     5.15
buffer busy waits                                  42,376       2,541     3.40
          -------------------------------------------------------------
```

可以看出，db file sequential read 是等待事件的第一位事件，占整个等待事件的 60%以上，并且平均每次等待的时间为 69ms。这是一个典型的 I/O 存在性能瓶颈的例子，通过 STATSPACK 报告文件 I/O 性能分析数据，可以进一步检查到底哪些文件出现了问题：

```
INDEX_SPACE_OTHER        /dev/vg10xp/rls_2g_vol05
         9,171       2   52.2     1.0        7,911
```

```
                /dev/vg10xp/rls_2g_vol06
  8,016       2   22.8      1.0        8,292
                /dev/vg10xp/rls_2g_vol07
  7,567       2    9.8      1.0        8,058
                /dev/vg10xp/rls_2g_vol08
  5,456       1   46.7      1.0        6,180
                /dev/vg10xp/rls_2g_vol09
  5,925       2   57.3      1.0        6,265
                /dev/vg10xp/rls_2g_vol10
 10,426       3  147.7      1.0        6,867
                /dev/vg10xp/rls_2g_vol11
  6,071       2  130.0      1.0        5,638
                /dev/vg10xp/rls_2g_vol12
  1,738       0  213.9      1.0            3
                /dev/vg10xp/rls_2g_vol13
  3,334       1  114.4      1.0        3,596
                /dev/vg10xp/rls_2g_vol14
 10,758       3  213.6      1.0       13,657
                /dev/vg10xp/rls_2g_vol15
 15,624       4  226.9      1.0       15,736
                /dev/vg10xp/rls_2g_vol16
 14,421       4  206.2      1.0       17,136
                /dev/vg10xp/rls_2g_vol17
 13,677       4  229.9      1.0       13,819
                /dev/vg10xp/rls_2g_vol18
 10,554       3  212.6      1.0       11,828
                /dev/vg10xp/rls_2g_vol19
  7,510       2  200.6      1.0        9,965
                /dev/vg10xp/rls_2g_vol20
  6,134       2  198.3      1.0        7,783
                /dev/vg10xp/rls_2g_vol21
  5,753       2  217.6      1.0          839
                /dev/vg10xp/rls_2g_vol22
  4,908       1  202.3      1.0          575
                /dev/vg10xp/rls_2g_vol23
 16,341       4  236.4      1.0        6,470
```

可以看出/dev/vg10xp 上部分文件的平均读响应时间超过了 200ms，存在严重的性能问题。通过验证，/dev/vg10xp 是 c55t14d1 上的逻辑卷。

优化小技巧 STATSPACK 报告之 I/O 问题分析

I/O 性能分析是 STATSPACK 分析中的重要部分。对于 I/O 的分析可以基于两个方面，第一个方面是在 Wait Events for DB 中，比如下面的数据：

```
                                                                   Avg
                                                     Total Wait   wait     Waits
Event                               Waits   Timeouts   Time (s)   (ms)      /txn
---------------------------- ------------ ---------- ---------- ------ --------
db file sequential read        14,409,809          0    424,347     29     93.6
SQL*Net more data to client     1,187,633          0      8,024      7      7.7
buffer busy waits                 212,482          0      5,888     28      1.4
```

```
db file scattered read            143,778          0      3,154     22      0.9
```

从上面来看，db file sequential read（29ms）和 db file scattered read（22ms）的等待时间都很高，说明 I/O 存在明显的性能问题。对于不同的系统，这些值的正常范围都不太相同，可以通过长期的积累，形成基线数据。不过，一般来说超过 10ms 就应该关注了，超过 20ms，I/O 子系统就可能存在问题了。从本例来看，系统的 I/O 性能存在一定的问题。为了确定猜测，可以进一步检查文件 I/O 的详细信息：

```
File IO Stats for DB: HCSP  Instance: hcsp  Snaps: 21 -33
->ordered by Tablespace, File

Tablespace                Filename
------------------------ ----------------------------------------------------
                 Av      Av     Av                    Av        Buffer Av Buf
         Reads Reads/s Rd(ms) Blks/Rd       Writes Writes/s      Waits Wt(ms)
-------------- ------- ------ ------- ------------ -------- ---------- ------
HAIER_TEST_DATA          /dev/vgrac/rhaier_test_data02
           132       0  163.6     2.5           65        0          0

HCSP_AI_DATA             /dev/vgrac/rHCSP_AI_DATA_01
         1,363       0   19.1     1.0        1,349        0          0

HCSP_AI_INDEX            /dev/vgrac/rHCSP_AI_INDEX_01
         4,649       1   17.5     1.0        3,614        0          2    0.0

HCSP_BASE_DATA           /dev/vgrac/rhcsp_base_data01
       329,857      38   14.3     2.8       77,415        9     33,510   11.2

HCSP_BASE_INDEX          /dev/vgrac/rhcsp_base_index01
        72,557       8   14.7     1.0          419        0        111    3.2

HCSP_COMM_DATA           /dev/vgrac/rhcsp_comm_data01
         7,789       1  112.1     2.7          692        0      3,884   87.5
```

从上面的数据可以看出，大多数的文件访问的平均读响应时间都超过 20ms。基本上我们可以得出结论，系统的 I/O 存在问题。

6月18日 准备收工

早上 8 点多和小齐会合后一起来到浑南的省公司。老方本来也要来，只是被吉林的事情绊住了。昨天晚上我和老于都没睡好，老于的眼睛红红的，我也一直打着哈欠。看样子小齐也没睡好，脸色有点苍白。我开玩笑说女孩子别有太多心思，老得快。可能大家心里都有些忐忑，这个玩笑也没有达到应有的效果，小齐也只是朝我撇撇嘴，表示收到。走进会议室之前，我打了个电话给小孙，问他昨天模块上线的情况以及现在系统的情况。小孙告诉我，昨天晚上他们原计划上线的 14 个模块，最终上线了 6 个，其他模块领导临时决定暂时不上了。因为新版本升级后，这几个模块要大改，目前上线意义不大。小孙不会监控系统，所以无法确定系统到底怎么样，不过从应用那边看，几个应用系统的性能都不错，和昨天的感受差不多。我让小孙和下面的营业厅联系一下，看看今天的系统情况，有事立即通知我。到了会议室，张工正在安排投影，我看他的笔记本通过无线连在 DCN 网上，就通过张工的笔记本连到服务器上，查看了 3 台服务器的状态。从数据库和操作系统的性能指标上看，都比昨天略好一些，看来昨天上线的 6 个模块表现还不错。大家看到这个情况，心里踏实多了。

孙主任临时有点事，晚了 10 多分钟才到了会场。之前和张工他们聊了聊目前系统的情况，他们对优化效果都感到比较满意。张工问我系统性能提升的比例大概有多少，是不是能够超过 30%的预期。我说现在还没做评估，还不好说，不过以我的经验，超过一倍应该没问题，远远高于 30%的预期。孙主任一到现场首先向我们表示祝贺，他来之前也让人和几个营业厅确认过了，今天系统状态十分好，好几个常用模块都比以前快很多。

今天的会议比较正式，首先老于向孙主任他们汇报了到目前为止所做的优化项目以及达到的效果。然后开发商也汇报了应用修改的情况，按照开发商的统计，他们对我们提出的应用修改，已经 100%完成，超过 90%的模块都已经上线，剩下没上线的是从业务角度考虑，准备在下一个小版本升级时一起做。

由于优化目的基本达到，所以这次小结会开得比较轻松，集团公司的领导也通过电话会议参加了讨论。会上大家一致同意本次优化到此为止，不再进行下一次微调。随后就是后续的一些安排。根据后面的安排，我过几天就可以回家了，老于还要再观察几天，下周也可以撤离现场。7 月 1 日，大家回到沈阳，进行账期数据的收集，以便于完成最终的性能评估报告。7 月 7 日至 7 月 10 日进行验收。随后安排一次北方公司的性能优化研讨会，在该研讨会上，以沈阳的案例介绍系统性能优化的方法与技术，对北方公司的 DBA 进行一次全面的培训。在这段时间里，我们

需要初步完成《系统性能分析报告》、《系统性能评估报告》、《优化方案》等正式文档。

会议结束后，我和老于先下楼。老于的烟瘾又犯了，出了门马上把烟点上。小齐还在和孙主任确认一些事情，我们在楼下等她。我也向老于要了一支烟点上，绷得很紧的神经终于松弛了下来。就这样结束了，心里放下了一块石头，反而觉得空落落的。

中午和小齐一起在北站旁边的“新洪记”吃了顿饭。心情高兴，多喝了几杯，下午就直接和老于一起回到酒店。想着下午到中街去买点沈阳土特产。老于建议晚上去刘老根大舞台看场二人转。我们就住在大舞台边上，一直也没心思去看，老于这个建议马上得到了我的响应。打了个电话给酒店前台，问问能不能帮助订票。忙了半天，买了两张 200 元的票，原价 180 元的，不过是二楼的，离舞台比较远。

今天下午接到了万隆的李科长的电话，问我有没有时间，帮他们的系统做一个调优，经费已经申请下来了，总共 15 万，他们出一半，集成商出一半，合同可以随后走流程，只要我们尽快进场做就可以了。万隆的系统问题很大，集成商在做软件开发时底层设计存在问题，导致每年 7 月、8 月、9 月这 3 个月业务高峰期时年年出问题。我一年前和李科长讨论过，希望大家一起坐下来，把底层的应用架构和数据字典调整好，这样就可以一劳永逸地解决问题。当时我提出的价格就是 15 万，不过当时他们觉得价格太贵，没地方出这笔经费，这件事就作罢了。没想到李科长还记着这个事情，而且运作出了这个项目。经过一年的运行，我估计万隆的系统比一年前更加恶化了。这个优化项目干不好，可能会适得其反。而且沈阳的项目刚刚结束，我也想好好休息一下，不想连续去碰硬骨头，所以我的初步想法就是推掉。不过李科长一副死缠烂打的架势，搞得我也不好彻底推掉，只是说先给我开个 VPN，我先安排人上去看看，了解一下能不能搞定再做决定。

放下电话后，和南京的阿罗联系了一下，让他有空 VPN 上去看看系统的情况。我再三嘱咐阿罗，这个系统问题很大，我们还没决定是不是要接这个项目，因此登上去后，只能看，不能动任何东西。并且这个系统十分关键，千万不能出问题，VPN 账号不要透露给公司的其他人，只有他一个人知道就行了。嘱咐完阿罗，老于的电话就进来了。两个人一起喝了点小酒就去刘老根大舞台看二人转。小时候也经常听二人转，不过今天看赵本山的弟子演出的二人转，还是感到十分过瘾。演出刚开始时报幕员说赵本山今天也在沈阳，如果能安排得开，会亲自表演一个节目。虽然最后由于时间错不开，赵本山只是出面打了个招呼就走了，不过看到这么一场演出，感觉 200 块钱花得还是很值的。

优化小技巧 STATSPACK 报告分析之 DB Cache2

DB Cache 的性能对于数据库的性能关系很大。从文件头的信息，我们可以对 DB Cache 的情况有个初步的认识：

```
Instance Efficiency Percentages (Target 100%)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
            Buffer Nowait %:   99.90       Redo NoWait %:  100.00
            Buffer  Hit   %:   92.76    In-memory Sort %:  100.00
```

```
             Library Hit   %:   97.56        Soft Parse %:   95.07
          Execute to Parse %:    0.89         Latch Hit %:   98.33
Parse CPU to Parse Elapsd %:   20.18     % Non-Parse CPU:   87.69
```

从上面的数据可以看到，Buffer Hit的比例是92.76%。作为一个OLTP系统，这个命中率不算太高，可能存在DB Cache过小的问题。另外，Buffer Nowait的比例，是99.9%，虽然这个比例还是比较高的，不过在后面还是要关注一下buffer busy waits大的情况。

下面进一步观察DB Cache的信息：

```
Buffer Pool Statistics for DB: BSS97  Instance: BSS97  Snaps: 40115 -40116
-> Standard block size Pools  D: default,  K: keep,  R: recycle
-> Default Pools for other block sizes: 2k, 4k, 8k, 16k, 32k

                                                            Free    Write  Buffer
     Number of Cache      Buffer    Physical    Physical  Buffer Complete    Busy
P      Buffers Hit %        Gets       Reads      Writes   Waits    Waits   Waits
--- ---------- ----- ----------- ----------- ---------- ------- --------- ------
D      956,500  92.8 611,070,386  44,260,765    333,301       0         1 317,535
          -------------------------------------------------------------
```

从上面的数据可以看到，Write Complete Waits的等待不是0，说明有会话在等待DBWR将脏块写入数据文件，然后释放。如果这个统计值和Free Buffer Waits的值不是0，那都有可能说明DB Cache过小。另外要关注Buffer Busy Waits的数量。在1小时内有317 535次Buffer Busy Waits，看样子这个系统的热块冲突还是挺严重的。加大DB Cache也有助于减少Buffer Busy Waits的数量。从上面的信息可以得出结论，这个系统的DB Cache确实可能有点小了，该如何调整DB Cache呢？从Oracle 9i开始，STATSPACK报告中增加了Buffer Pool Advisory。通过这部分数据，可以确定DB Cache加大后的效果怎么样：

```
        Size for  Size      Buffers for  Est Physical          Estimated
P   Estimate (M) Factor        Estimate   Read Factor     Physical Reads
--- ------------ ----- ---------------- ------------- ------------------
D            144    .1           17,865          6.76     10,064,518,738
D            288    .2           35,730          4.91      7,307,758,071
D            432    .3           53,595          4.24      6,311,657,227
D            576    .4           71,460          2.67      3,973,996,431
D            720    .5           89,325          2.32      3,449,994,052
D            864    .6          107,190          2.01      2,994,522,015
D          1,008    .7          125,055          1.72      2,558,135,498
D          1,152    .8          142,920          1.50      2,229,794,150
D          1,296    .9          160,785          1.27      1,891,321,676
D          1,440   1.0          178,650          1.08      1,602,214,384
D          1,504   1.0          186,590          1.00      1,488,273,940
D          1,584   1.1          196,515          0.80      1,184,885,252
D          1,728   1.1          214,380          0.70      1,046,836,171
D          1,872   1.2          232,245          0.63        931,974,901
D          2,016   1.3          250,110          0.56        837,755,990
D          2,160   1.4          267,975          0.51        761,325,613
D          2,304   1.5          285,840          0.47        695,515,934
D          2,448   1.6          303,705          0.43        639,509,106
D          2,592   1.7          321,570          0.39        586,920,905
```

```
D          2,736   1.8          339,435          0.35        525,299,156
                ------------------------------------------------------------
```

目前的 DB Cache 的大小是 1 504MB，如果目前系统产生的物理读的数量是 1.0 的话，那么 DB Cache 加大到 1 728MB 时，物理读的数量会下降 20%，I/O 减少的幅度是十分明显的。DB Cache 增加到 2 304MB 时，物理读的数量是目前的 47%。DBA 可以根据目前系统物理内存的情况，选择一个适当的 DB Cache 的大小。

6月19日 ORA-1591

这几天系统状态都很不错，我们这个优化项目也接近尾声了。能够取得如此成功，是大家以前想都不敢想的事情。老于这几天简直乐开了花，连烟都少抽了不少。这几天我们去现场的时间都比往常推迟了，一般 9：00 才出门，9：30 左右到办公室，晚上也早早收工，避开下班高峰期。昨天晚上看一本小说，睡得晚了一点，今天我出门比平时更晚，到办公室时老于也不在。电脑包放在办公桌上还没打开，小孙就说老于刚到办公室时，接了一个电话就出去了。我想反正也没啥事情，就打开电脑，开始上网聊天。这段时间项目吃紧，连上 QQ 聊天的时间都少了。

今天持续了前几天的好状态，几台服务器的 CPU 使用率都在 60%左右，I/O 性能也很不错，avserv 基本上在几毫秒。从小孙那边反馈回来的信息也不错，这几天关于性能的投诉基本上没有了，BBS 上的牢骚也少了不少。

快中午时老于还没有回来，我打了一个电话问他中午回不回来，他说不回来了，于是准备一个人去吃午饭。刚走到楼梯口，小孙追了出来，说："老白，先别走，系统出了个问题，帮我看看。"

我回到办公室，问他什么问题，小孙说有个业务，一跑就报错，而其他业务都没问题。我连忙查看了一下警告日志，发现警告日志里在不停地报错：

```
SMON: about to recover undo segment 119
ORACLE Instance topcs2 (pid = 11) - Error 1591 encountered while recovering tran
saction (119, 18) on object 2309045.
Errors in file /u2/oracle/ora92/rdbms/log/topcs2_ora_2899.trc:
ORA-01591: lock held by in-doubt distributed transaction 108.28.46269
SMON: mark undo segment 119 as needs recovery
```

一看这个错误号 ORA-1591，就知道这是典型的分布式事务故障导致的问题。当一个分布式事务由于某种原因在 prepare 阶段出现故障时，会导致本地相关的资源被锁定，直到分布式事务被回退或者提交为止。我问小孙故障出现前发生了什么，小孙说刚才银行收费的接口出了问题死掉了，他刚才重启了这个程序，其他都正常，就是这个业务总报错。

我问小孙这个接口模块是不是 Tuxedo 的，小孙说是 Tuxedo 6.5 的应用。这种 XA 应用出问题导致 ORA-1591 的问题也算是常见问题了，我马上查找 119 号回滚段里是不是有死事务：

```
SELECT KTUXEUSN, KTUXESLT, KTUXESQN,
               KTUXESTA Status, KTUXECFL Flags
  FROM x$ktuxe
```

```
WHERE ktuxesta!='INACTIVE'
     AND ktuxeusn=119;
```

查询结果是没有任何记录。我一看原来 in-doubt 的分布式事务是 108.28.46269，于是马上查询 108 号回滚段里是不是有死事务：

```
SELECT KTUXEUSN, KTUXESLT, KTUXESQN,
               KTUXESTA Status, KTUXECFL Flags
  FROM x$ktuxe
WHERE ktuxesta!='INACTIVE'
     AND ktuxeusn=108;
 KTUXEUSN   KTUXESLT   KTUXESQN STATUS           FLAGS
---------- ---------- ---------- ---------------- ------------------------
       108         28      46269 PREPARED         SCO|COL|REV|DEAD
```

确实存在一个 PREPARED 状态的死事务。于是马上检查了一下 pending_trans$表：

```
SELECT local_tran_id, global_tran_fmt, global_Oracle_id,
   global_foreign_id, state, status, heuristic_dflt,
          session_vector, reco_vector,
        global_commit#
        FROM pending_trans$;
LOCAL_TRAN_ID GLOBAL_TRAN_FMT GLOBAL_ORACLE_ID GLOBAL_FOREIGN_ID STATE STATUS
HEURISTIC_DFLT SESSION_VECTOR RECO_VECTOR GLOBAL_COMMIT#

70.6.108873 306206 TOPCS.c7dd20c6.70.6.108873  forced rollback P  00000001 00000001
1486865870
96.42.84009 306206 TOPCS.c7dd20c6.96.42.84009  forced rollback P  00000001 00000001
1286932454
85.10.101067 306206 TOPCS.c7dd20c6.85.10.101067  forced rollback P  00000001 00000001
1487167659
9.35.29156 306206 TOPCS.c7dd20c6.9.35.29156 collecting P 00000001 00000001 1672793351
80.8.132177 306206 TOPCS.c7dd20c6.80.8.132177  forced rollback P  00000001 00000001
1679427495
18.32.162778 306206 TOPCS.c7dd20c6.18.32.162778  forced commit P  00000001 00000001
64.21.136442 306206 TOPCS.c7dd20c6.64.21.136442  forced commit P  00000001 00000001
73.11.124822 306206 TOPCS.c7dd20c6.73.11.124822  forced rollback P  00000001 00000001
1731227073
63.29.148558 306206 TOPCS.c7dd20c6.63.29.148558  forced commit P  00000001 00000001
```

发现并不存在 108.28.46269 这个事务。看样子是系统字典表出现问题了，导致这个分布式事务无法被自动回退。首先看看这个事务锁定的对象是什么：

```
SELECT NAME FROM OBJ$ WHERE OBJ#=2309045;
NAME
-------------------------
TPTB_IDX
```

这是一个索引，基表是 T_TPTB。对这张表进行全表扫描，看看会发生些什么：

```
SELECT /*+ full(a) */ count(*) FROM T_TPTB a
ORA-01591: lock held by in-doubt distributed transaction 73.11.124822
```

这张表被另外一个分布式事务 73.11.124822 锁定了。查看一下 73 号回滚段的情况：

```
SELECT KTUXEUSN, KTUXESLT, KTUXESQN,
```

```
               KTUXESTA Status, KTUXECFL Flags
  FROM x$ktuxe
WHERE ktuxesta!='INACTIVE'
     AND ktuxeusn=73;
  KTUXEUSN   KTUXESLT   KTUXESQN STATUS           FLAGS
---------- ---------- ---------- ---------------- ------------------------
        73         11     124822 PREPARED         SCO|COL|REV|DEAD
```

执行 ROLLBACK FORCE 73.11.124822 时出现了错误：

```
SQL>ROLLBACK FORCE '73.11.124822'
ERROR at line 1:
ORA-02058: no prepared transaction found with ID 73.11.124822
```

在 pending_trans$里查看目前这个事务的状态不是 PREPARED,因此需要首先在 pending_trans$里将这个事务的状态设置为 prepared：

```
UPDATE pending_trans$SET        STATE='prepared',
        STATUS ='P'
WHERE local_tran_id='73.11.124822';
COMMIT;
```

执行了这个语句后，再次执行 ROLLBACK FORCE '73.11.124822'还是报错：

```
SQL> ROLLBACK FORCE '73.11.124822';
ROLLBACK FORCE '73.11.124822'
*
ERROR at line 1:
ORA-01591: lock held by in-doubt distributed transaction 108.28.46269
```

看样子这个事务被我们最初发现的那个事务锁住了。这两个事务一个锁住了表，一个锁住了索引，所以只能看看能不能强制提交了：

```
SQL>UPDATE pending_trans$ SET        STATE='prepared',
        STATUS ='P'
where local_tran_id='73.11.124822';
SQL>commit;
SQL>COMMIT  FORCE '73.11.124822';
```

使用这个命令，这个事务被成功地强制提交了。下面来处理 108 回滚段的那个事务。由于这个事务在 pending_trans$中缺少记录，所以首先将这个事务清理掉：

```
SQL>exec dbms_transaction.purge_lost_db_entry( '108.28.46269' )
```

然后手工插入相关记录：

```
SQL>alter system disable distributed recovery;
SQL>insert into pending_trans$ (
        LOCAL_TRAN_ID,
        GLOBAL_TRAN_FMT,
        GLOBAL_ORACLE_ID,
        STATE,
        STATUS,
        SESSION_VECTOR,
        RECO_VECTOR,
        TYPE#,
```

```
        FAIL_TIME,
        RECO_TIME)
    values( '108.28.46269',
        306206,
        'XXXXXXX.12345.1.2.3',
        'prepared','P',
        hextoraw( '00000001' ),
        hextoraw( '00000000' ),
        0, sysdate, sysdate );

SQL>INSERT INTO pending_sessions$
    VALUES( '108.28.46269',
        1, hextoraw('05004F003A1500000104'),
        'C', 0, 30258592, ",
        146
      );

SQL>Commit;
```

然后再次强制提交：

```
SQL> COMMIT FORCE '108.28.46269';

Commit complete.
```

完成提交后，小孙再次检查了一下那个应用，发现问题解决了。

今日点评

今天老白处理的这个案例是一个十分典型的分布式事务失败导致的问题。这类问题出现的机会很大，特别是在使用了 Tuxedo、CICS 和透明网关等的情况下。如果碰到类似问题，该如何处理呢？老白在多年的工作经验中有以下的经验和大家分享。首先我们来看看 Oracle 官方对 ORA-01591 的描述：

```
ORA-01591: "lock held by in-doubt distributed transaction %s"
        Cause: Trying to access resource that is locked by a dead
               two-phase commit transaction that is in prepared state.
       Action: DBA should query the pending_trans$ and related tables,
               and attempt to repair network connection(s) to
               coordinator and commit point. If timely repair is not
               possible, DBA should contact DBA at commit point if
               known or end user for correct outcome, or use heuristic
               default if given to issue a heuristic commit or abort
               command to finalize the local portion of the
               distributed transaction.
```

总的来说，ORA-01591 的产生原因是分布式事务失败，失败的原因很多，比如网络问题、XA 资源管理器存在 bug 等。一旦分布式事务失败，如果本地事务中有一个事务正处于活跃状态，那么该事务相关的数据就会被锁定（无论读写都会被锁定）。如果访问这个事务关联的数据，就会报 ORA-01591。一般情况下，ORA-01591 可以自动解开，SMON 会在一定时间周期内检查

DBA_2PC_PENDING，找出需要回退的事务，并进行自动恢复。这里就有几个问题，由于分布式事务超时判断以及 redo 处理周期的关系，一般来说事务自动恢复的时间为 1 分钟以上，较长的可以达到 5～10 分钟，这可能会对生产系统造成比较大的影响。为了加快解锁，可以手工处理。这时可以使用 ROLLBACK FORCE 或者 COMMIT FORCE。

有时候由于分布式事务恢复出现故障，会出现数据字典不一致，此时该分布式事务就无法正常解除，需要手工干预来处理。具体的操作步骤如下所示。

(1) 检查分布式事务的状态：

```
SELECT LOCAL_TRAN_ID, GLOBAL_TRAN_ID, STATE, MIXED, HOST, COMMIT#
FROM DBA_2PC_PENDING
WHERE LOCAL_TRAN_ID = '报错的本地事务号'
```

(2) 检查与分布式事务相关的其他节点的情况：

```
SELECT LOCAL_TRAN_ID, IN_OUT, DATABASE, INTERFACE
FROM DBA_2PC_NEIGHBORS;
```

(3) 检查本地回滚段：

```
SELECT KTUXEUSN, KTUXESLT, KTUXESQN, /* 事务号 */
    KTUXESTA Status,
    KTUXECFL Flags
FROM x$ktuxe
WHERE ktuxesta!='INACTIVE'
    AND ktuxeusn= 回滚段编码
```

最常见的情况是 DBA_2PC_PENDING 中存在一条 prepared 状态的记录，在回滚段中也发现一个是 prepared 状态的活跃事务，但是事务有 DEAD 标志。这种情况下，可以根据事务号进行 COMMIT FORCE 或者 ROLLBACK FORCE 操作。

另外一种情况是 DBA_2PC_PENDING 中也有记录，但是状态和回滚段中的状态不同，这时可以手工修改 sys.pending_trans$中的状态值，然后再进行 ROLLBACK FORCE/COMMIT FORCE 操作。

如果在 DBA_2PC_PENDING 中找不到记录，那么要检查 sys.pending_trans$、sys.pending_sessions$和 sys.pending_sub_sessions$。如果记录丢失，就可以手工插入记录，然后进行 ROLLBACK/COMMIT FORCE：

```
insert into pending_trans$ (
    LOCAL_TRAN_ID,
    GLOBAL_TRAN_FMT,
    GLOBAL_ORACLE_ID,
    STATE,
    STATUS,
    SESSION_VECTOR,
    RECO_VECTOR,
    TYPE#,
    FAIL_TIME,
    RECO_TIME)
values('transid',
```

```
        306206,        固定值/
        'XXXXXXX.12345.1.2.3', 随意取值
        'prepared','P',           表示 prepared 状态
        hextoraw( '00000001' ),  /* 常量                              */
        hextoraw( '00000000' ),  /*                                   */
        0, sysdate, sysdate );

insert into pending_sessions$
values( '事务号',
        1, hextoraw('05004F003A1500000104'),
        'C', 0, 30258592, '',
        146
    );

commit;
```

(4) 做完 ROLLBACK FORCE/COMMIT FORCE 后，如果 DBA_2PC_PENDING 中的记录未清除或者 DBA_2PC_PENDING 中有记录，但是回滚段中已无记录，那么可以使用下列代码将 DBA_2PC_PENDING 中的记录清除掉：

```
ALTER session SET "_smu_debug_mode" = 4;
//如果是 Oracle 9i，使用 AUM 需要设置，否则后面会出错
commit;
exec dbms_transaction.purge_lost_db_entry( '事务号' )
```

我们实际碰到的情况可能要比这里介绍的复杂得多，更详细的信息可以参考 Metalink 文档 401302.1。

6月20日 好奇心惹的祸

昨天晚上订好了今天中午的航班，本来想早上睡个懒觉，不过不到 8：00 就醒了。上网浏览了一下新闻，就下去吃早餐。在餐厅刚刚挑了一盘子点心，正在倒牛奶，手机就响了。作为一名 DBA，最怕的是半夜手机响。这种刚刚上班时手机就响，我预感要有不好的事情发生了。

掏出手机一看，是山东的电话，就知道肯定是李科长打来的电话。他焦急地问我，是不是我们昨天晚上做了什么，怎么今天一上班，系统就像死了一样。我说："昨晚我们只是上去看看，并没有操作什么，我马上连上去看看吧。"说完我马上回到房间。路上我给阿罗打了一个电话，问他昨天晚上做了什么。阿罗接电话时还睡眼朦胧的，他告诉我昨天晚上他上去分析了一下，发现有几个 SQL 执行计划不正确，他觉得对其中一张大表重新分析一下应该能够解决问题，所以晚上他对那张表做了 40%采样的分析，并且对所有倾斜字段分析了柱状图。那个分析脚本到现在还没有完成。通过和阿罗的电话，我可以百分之百地确定，今天早上系统出现的问题和阿罗这个分析是相关的。

我马上叫阿罗停掉那个分析，然后我通过 VPN 连上了数据库。阿罗做分析的那张表大概有 80 多吉字节，不过这张表是个分区表，每个季度一个分区，表上面有 11 个索引，现在是 8：10，如果要在 8：30 全国营业网点上班前完成分析，那么只有 20 分钟可用。我立即写了两个脚本，对最近的两个分区（每个分区包含一个季度的数据）做一个 10%采样的分析，只分析表，不分析索引和柱状图。8：27，分析终于完成了。8：35，系统的 CPU 终于降低到 70%左右，系统基本上可以正常运行了。看到系统基本恢复正常，我将剩余的几个分区做了一个 10%的采样，并将所有索引做了 10%的采样分析。做完这些，已经将近 10：00 了。通过 OEM 看到目前系统十分繁忙，10：00 开始是业务高峰，这个系统每天的业务高峰是 10：00～11：00 以及下午的 3：00～4：00。CPU 使用率在 85%～90%，I/O 繁忙程度在 40%左右，物理内存使用率已经超过 90%。从目前系统的状态看，今天系统不会有太大的问题了。

我马上给李科长打了个电话，说从目前情况看，今天应该没问题，今天晚上我会把这张表重新分析一下，整个分析过程会在凌晨开始做，四五点钟结束，应该是比较安全的。我说的过程中，李科长一句话也没有说。等我说完，李科长说："老白，这回你的弟兄给我捅了这么大一个篓子，你就别想跑了。优化这件事，你必须接下来，否则今天早上这件事，我就可以让公司给你发律师函让你赔偿。"我说："李科长，我们也算老朋友了，不能这么无耻吧，你们这个系统优化风险太大，好多团队都不肯接你这个活。最近我们接了几个难啃的骨头，短时实在是没精力来干这个活了"。

说实在的，如果不是沈阳的事情还没有完全结束，我还是很想做李科长这个项目的。这种优化项目非常具有挑战性。不过，目前这个项目的风险指数十分高，如果真要做，肯定要全力投入才行。而且时间也十分紧迫，离下个月的业务高峰只有不到半个月的时间，要在这么短时间内有所作为，难度确实比较大。这个系统是全国性的服务系统，一旦出现问题，后果十分严重。就算我这个比较爱涉险的人，对这个活也不敢轻易答应。

李科长这回仍旧像牛皮糖一样难以甩掉，无奈之中我只好答应接下这个活，但是我说不能保证在这段时间全部投入到这个项目中，因为沈阳那边还有一个研讨会，大概会占用我一周的时间。李科长马上说："没关系，你不能到现场，远程关注一下就行，你把老熊派过来就行了。"老熊原本还有别的事情，暂时无法脱身，被李科长这么一说，也就没办法了。李科长听到了我的承诺，心满意足地放下了电话，我却又要头疼了，老熊被李科长抢过去了，他的缺怎么补呢？算了，走一步算一步吧。

放下电话，先给老熊去了个电话，让他准备准备，明天就出发。想到阿罗今天捅的这个篓子，气不打一处来，立即打电话给阿罗。阿罗已经知道自己今天捅了个小篓子，所以一接电话就说："老白，又给你添麻烦了。"我说："麻烦不是一般的大，李科长这种人是招惹得起的吗。对于这个项目，本来我的想法是能推就推，大不了给他提几条免费建议就完了，这回被他粘上了。你也算个老 DBA 了，铁的纪律都能违反？"

阿罗也是一肚子委屈，他也知道这个系统不能轻易动手，不过昨天晚上他分析了半夜，觉得自己已经找到了问题的根源，分析一下表，可能就搞定了，所以他禁不住诱惑，做了那个操作。我以前就和阿罗说过，他最大的毛病是好奇心太强，作为一名 DBA，这是十分致命的。作为一名 DBA，最本分的做法是只做分内的事情，其他的能不做尽量不做，特别是没有百分之百把握的事情。以往的经验证明，经不住诱惑时做的操作绝大多数都会导致严重的后果。

一个不经意的操作把公司这段时间的安排全部打乱了。不过阿罗已经方寸大乱，这个时候也是多说无益。我和他通电话时夸大这个问题的严重性，目的也是想让阿罗长点记性，不要再犯类似的错误。虽然被迫接下了这个优化项目，不过有活干也不是件坏事，其他的项目安排我只能另外想办法了。

被这件事折腾了一上午，原本到中街买点土特产的计划也就泡汤了。匆匆忙忙退了房，打了个车就直奔机场。看样子最近又要跑一趟山东了。

优化小技巧 STATSPACK 报告分析之共享池分析

对共享池的分析首先应该从 Load Profile 开始：

```
Load Profile
~~~~~~~~~~~~                            Per Second       Per Transaction
                                   ---------------       ---------------
                  Redo size:            466,915.94             61,799.72
              Logical reads:             30,110.84              3,985.39
              Block changes:                185.44                 24.54
             Physical reads:              1,682.72                222.72
```

```
         Physical writes:              179.06              23.70
              User calls:              261.34              34.59
                 Parses:                90.72              12.01
            Hard parses:                18.04               2.39
                   Sorts:                9.48               1.26
                  Logons:                1.18               0.16
                Executes:              102.75              13.60
            Transactions:                7.56

  % Blocks changed per Read:    0.62    Recursive Call %:     35.95
 Rollback per transaction %:    5.48       Rows per Sort:   2737.16

Instance Efficiency Percentages (Target 100%)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
            Buffer Nowait %:   99.94       Redo NoWait %:    100.00
            Buffer  Hit   %:   94.77    In-memory Sort %:     99.97
            Library Hit   %:   93.73        Soft Parse %:     80.11
         Execute to Parse %:   11.70         Latch Hit %:     99.77
Parse CPU to Parse Elapsd %:   62.93     % Non-Parse CPU:     97.97
 Shared Pool Statistics        Begin   End
                               ------  ------
             Memory Usage %:   86.38   89.00
    % SQL with executions>1:   45.76   46.74
  % Memory for SQL w/exec>1:   60.07   57.53
```

上面加粗的内容都是和共享池有关的。从上面的数据看，每秒 Parses 的数量是 90.72，其中 Hard parses 的数量是 18.04，Soft Parse 的比例为 80.11%，说明系统的硬分析较为严重，对共享池的压力较大。从 Library Hit 来看，93.73%的命中率对于一个 OLTP 系统来说还是略低一些。从这些统计数据来看，这个系统可能存在一定共享池问题。在后面的报告中我们需要仔细分析。通过闩锁情况，也可以看出共享池是否存在问题：

```
                                            Pct    Avg   Wait                 Pct
                                Get         Get   Slps   Time       NoWait NoWait
Latch                       Requests       Miss  /Miss    (s)     Requests   Miss
------------------------ -------------- ------ ------ ------ ------------ ------
library cache                 3,937,180    5.1    0.0      4      121,176    0.5
library cache load lock             538    0.0             0            0
library cache pin             1,731,246    0.4    0.0      0            0
library cache pin alloca      1,166,244    0.1    0.0      0            0
row cache enqueue latch       4,330,937    0.3    0.0      0            0
row cache objects             4,379,188    0.2    0.0      0           39    0.0
shared pool                   2,218,835    0.5    0.0      0            0
```

从上面的数据可以看出，library cache 闩锁的丢失率比较高，这和硬分析较高有关，但不能说明共享池就过小，因为 row cache 和 shared pool 相关的闩锁丢失率在正常的范围。可以继续查看共享池的相关信息：

```
Dictionary Cache Stats for DB: NEVA2DB  Instance: neva2db  Snaps: 18596 -18597
->"Pct Misses"  should be very low (< 2% in most cases)
->"Cache Usage" is the number of cache entries being used
->"Pct SGA"     is the ratio of usage to allocated size for that cache
```

```
                                Get    Pct    Scan   Pct      Mod      Final
Cache                      Requests   Miss    Reqs  Miss     Reqs      Usage
------------------------- ------------ ------ ------- ----- -------- ----------
dc_constraints                  195   34.9       0            195          6
dc_files                      4,752    0.0       0              0        176
dc_histogram_defs           249,043    0.1       0              0      6,728
dc_object_ids             1,071,024    0.0       0             21        817
dc_objects                  152,381    0.1       0             68      1,325
dc_profiles                   2,118    0.0       0              0          1
dc_rollback_segments         10,538    0.0       0              0        171
dc_segments                 303,970    0.0       0          3,857      1,387
dc_sequences                    333    0.0       0            333         12
dc_table_scns                    19  100.0       0              0          0
dc_tablespace_quotas          1,626    0.1       0          1,626          9
dc_tablespaces              154,730    0.0       0              0        151
dc_user_grants               18,339    0.0       0              0         13
dc_usernames                 54,441    0.0       0              0         11
dc_users                    164,074    0.0       0              0         14
          -------------------------------------------------------------

Library Cache Activity for DB: NEVA2DB  Instance: neva2db  Snaps: 18596 -18597
->"Pct Misses"  should be very low

                         Get  Pct        Pin        Pct               Invali-
Namespace           Requests  Miss     Requests     Miss     Reloads  dations
--------------- ------------ ------ -------------- ------ ---------- --------
BODY                     367    0.0            367    0.3          1        0
CLUSTER                  163    0.0            178    0.0          0        0
INDEX                    346    1.2            282    1.4          0        0
SQL AREA             163,152   19.3        590,480   10.9        849       70
TABLE/PROCEDURE      159,959    0.1        441,195    0.1         55        0
TRIGGER                2,848    0.0          2,848    0.0          0        0
          -------------------------------------------------------------
```

SQL AREA 的丢失率是 19.3，其他都很正常。SQL AREA 的丢失率较大和硬分析比例较高有关。一般来说，SQL AERA 的丢失率在 20%以内都是可以接受的。下面我们继续查看 Shared Pool Advisory：

```
Shared Pool    SP       Estd          Estd     Estd Lib LC Time
   Size for  Size  Lib Cache     Lib Cache   Cache Time   Saved  Estd Lib Cache
  Estim (M) Factr   Size (M)       Mem Obj    Saved (s)   Factr    Mem Obj Hits
----------- ----- ---------- ------------ ------------ ------- ---------------
        912    .6        804      105,936   14,768,878     1.0     326,665,094
      1,072    .7        964      125,874   14,768,910     1.0     326,693,626
      1,232    .8      1,124      150,371   14,768,938     1.0     326,717,820
      1,392    .9      1,284      175,241   14,768,967     1.0     326,742,860
      1,552   1.0      1,442      197,493   14,768,991     1.0     326,765,091
      1,712   1.1      1,601      218,503   14,769,010     1.0     326,781,693
      1,872   1.2      1,760      239,872   14,769,028     1.0     326,796,532
      2,032   1.3      1,919      261,536   14,769,040     1.0     326,808,261
      2,192   1.4      2,078      283,180   14,769,051     1.0     326,819,047
      2,352   1.5      2,237      305,581   14,769,064     1.0     326,831,187
      2,512   1.6      2,396      325,251   14,769,080     1.0     326,845,940
```

```
  2,672   1.7      2,556      342,048   14,769,097     1.0     326,861,145
  2,832   1.8      2,715      358,349   14,769,115     1.0     326,875,825
  2,992   1.9      2,874      374,365   14,769,133     1.0     326,891,217
  3,152   2.0      2,975      385,473   14,769,151     1.0     326,906,085
      ------------------------------------------------------------
```

从上面的图表来看，目前的共享池大小为 1 552MB，可以容纳的对象数量是 326 765 091。加大共享池后，容纳对象的数量增加有限。从上面的分析可以知道，虽然库 Cache 的命中率不足 95%，但是这主要由 SQL 硬分析较为严重导致，和应用有关，简单地通过加大共享池的大小作用不大。

7月20日 重回沈阳

算起来离开沈阳一个月了，6 月下旬离开沈阳时，优化实施的主体工作已经结束，只剩下部分 SQL 还没有上线。索引重建的工作我们安排在 6 月底账期开始之前进行，由留守的老于编写了一组存储过程，交给小孙实施。这个索引重建工作和表分析工作已经成为日常维护的常规项目，正式写入了开发商的维护工作指导书中，现场的应用维护人员会根据这个指导书，在固定的时间实施这些操作。

项目后期我们就没有再去沈阳了，因为自从 6 月 20 号大批 SQL 上线后，3 套系统的性能都有了很大幅度的提升，资源紧张的情况大大缓解，CPU、I/O、内存都处于比较安全的范围，系统响应时间提升都超过了一倍，系统总的吞吐量也有了较大幅度的提高。从全省各个营业厅的反馈情况来看，营业前台对系统性能的改善也十分认可。6 月底的账务处理异常顺利，经过优化后的账务处理程序，性能普遍提高了数倍。因此，6 月底的账务处理耗时比平时缩短了一倍以上，对此 IT 部和业务主管部门都比较满意。

7 月初时，老于给我发来了完整版本的《系统性能评估手册》，邮件正文只有胜过千言万语的 4 个字“不辱使命”。我理解老于发这封邮件时兴奋的心情，这种心情只能用这 4 个字来表达。于是我迫不及待地打开附件，想看看我们到底创造了一个什么样的奇迹，具体如下表所示。

性能指标	调整前	调整后	对比说明
响应时间/事务	3 867.39	1 000.74	平均每事务响应时间减少； 系统响应速度加快； 说明系统总体性能提高3.87倍
CPU时间/事务	2 125.48	355.36	平均每事务消耗的CPU时间缩短； 事务处理速度提高
等待时间/事务	1 741.91	645.38	平均每事务的等待事件时间减少； 系统等待时间降低，性能提高
物理读/秒	1 113.86	996.91	物理磁盘读减少 磁盘I/O降低，系统性能提高
事务数/秒	8 461	10 426	事务量高于调整前，系统负载提高
redo size/秒	12 807.29	25 323.94	每秒及每事务产生的redo大于调整前，说明系统负载及事务大小均高于优化前
redo size/事务	5 450.78	8 746.55	

从评估报告上可以很清晰地看到，系统响应时间提升了 3.87 倍，这在营业前台会有明显的

感受，这是最终用户真实感受的反映。从每个事务的 CPU 消耗来看，这部分的提升尤为明显，这部分成果主要来自于 SQL 优化以及表分区、索引重建等动作的效果。每个事务的等待时间也有了明显的减少，这是数据库整体性能提升的体现。上面这些数据确实是我在两个月前想都不敢想的，我们居然取得了这么辉煌的成功。最终能够获得这么漂亮的数据，和开发商最后阶段的成果是分不开的。如果不是最后关头那些 SQL 上线，我们也许能够获得 30%～50%的性能提升。想起来我和孙主任演的那出双簧的作用还是巨大的。

验收那天老于喝得大醉，最后是被张工他们抬回酒店的。实际上验收工作很简单，老于在 7 月初账期后回家休息了几天，回到沈阳时小齐就希望老于能够在 15 日之前完成验收工作。当老于忐忑不安地向孙主任提出验收请求时，没想到孙主任异常爽快，他说："验收只是个形式而已，这大半个月系统表现这么好，就是最好的验收。优化这种项目，谁说了都不算，系统说了才算数。你明天把验收所需要的材料带过来，我签个字就算验收了吧。不过正式盖章还需要几天，这是公司的流程，我也没办法。"

后来听张工说，签字那天，老于被张工他们放倒了，实际上也是老于自己把自己放倒的。酒是在滑翔小区边上的"海鲜大排档"喝的，开始喝啤酒，后来啤酒不过瘾了，就喝上二锅头了。大家都喝好了准备撤的时候，老于又突然提出去 K 歌，最后在 KTV，老于终于倒下了。这是一种宣泄，一种快乐的宣泄。每个人都有宣泄的方式，和老于不同，我在一个大项目结束时，喜欢一个人独自坐坐，稍微喝点红酒。

我把老于的邮件转给了老熊，老熊说老于已经和他通过电话了，他听说后也很兴奋，只是没赶上那顿酒，有点遗憾。我说就是你去了也是白给，沈阳的哥们喝酒和南方人不一样，基本上是去一个放倒一个。

昨天小齐打电话来说，21 日开始准备在沈阳开一个"系统性能优化研讨会"，邀请了北方公司各省的技术人员参加。会议地址安排在风景秀丽的棋盘山度假区，主要活动包括项目总结和技术培训。老于负责讲"Oracle 数据库管理"，我负责讲"Oracle 数据库性能优化实战技术"。课程内容我早就提交给小齐了，小齐和北方公司研究后觉得我的讲义里理论部分过多，因为参加活动的技术人员水平参差不齐，所以建议我降低难度，并且增加更多实战内容，最好把我们在沈阳做项目的案例揉到课程里，用实际案例来替代高深的理论。

今天一下飞机，就接到了小齐的短信，她已经在机场了，会务组准备了一辆中巴车，我正好能够赶上，否则自己一个人去那里还真有点不好找。棋盘山度假区位于沈阳市的水源地棋盘山水库腹地，虽然在沈阳生活过 10 年，但那个地方我还是头一次去。进了棋盘山风景区的大门，汽车又沿着水库跑了差不多 20 分钟才来到我们下榻的酒店。棋盘山里面的酒店都是临湖背山而建，沿着山坡一片不高的房子，给人感觉很舒服。

大家忙着签到领礼品，我看了一下礼品，里面有一套迷你音箱不错，就要了一套迷你音箱和一套修甲器。接待的小妹妹笑着对我说："你肯定是个好丈夫。"我说为什么，她说我是第一个选择修甲器的男士，领礼品还想着老婆的男人肯定是个好丈夫。想想也是，无意之中就做了一次好丈夫，看样子我从骨子里来说还是个不错的老公，回家要给老婆好好宣扬宣扬。我在来宾签到表

上看到了老于的名字，就让负责接待的小妹妹把我和老于安排在一个房间。一个多月不见了，很想和老于聊聊。

找老于的房间其实很容易，闻着烟味就可以找到了。我在房门口就闻到了一股浓烈的烟味，看样子老于正在房间里。虽然开着窗，屋里的烟味还是很浓，老于正坐在床上喷云吐雾。

我说老于，我们还是到湖边坐坐吧，这里烟味太浓，受不了。我和老于一起坐在湖边，一边抽烟一边听老于介绍我走后的一些事情。后来开发商配合得相当到位，我们提出的几十个 SQL 最后大多数都上线了，范总后来又来了一次。孙主任请老于和范总一起吃了顿饭，饭桌上孙主任把我们当时算计他的事情挑明了。范总说其实他也早就回过味来了，这件事肯定是个套，不过这个套他是自愿往里钻的，因为这件事是多赢的事情，没理由不接受。孙主任和范总都是海量，最后大家尽欢而散。

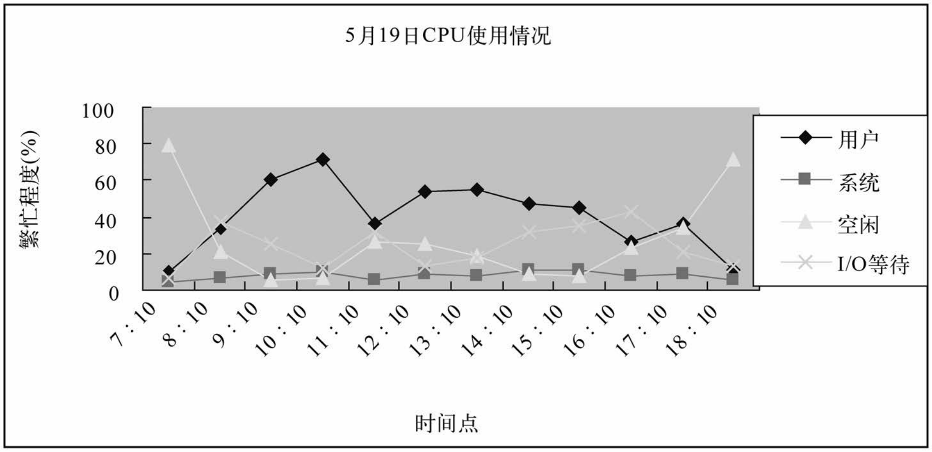
优化小技巧 如何评估优化效果

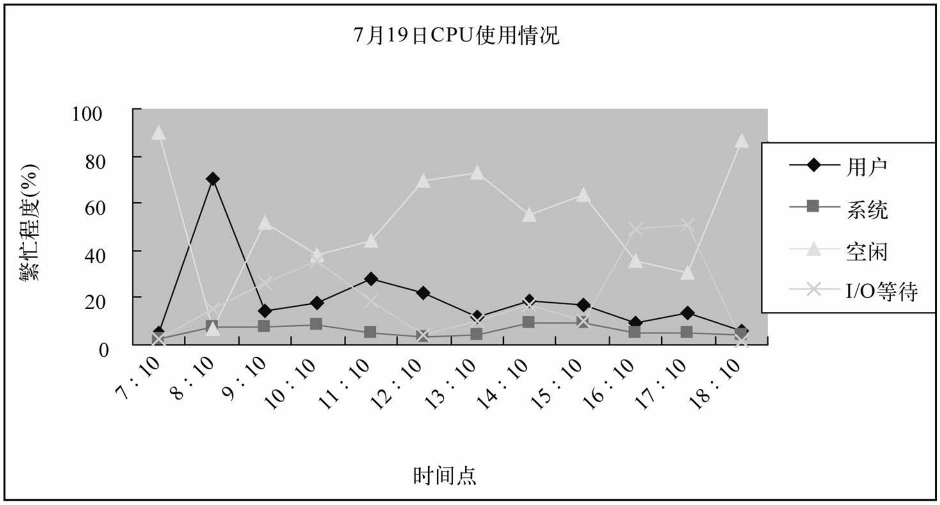
优化效果的评估不仅仅是优化实施者交差的手段。作为一个实施优化项目的人，在项目的每个阶段都需要去对优化效果进行评估，以便确定目前的优化是否达到了预期的目标，是否需要继续优化下去。如果是优化某个 SQL，那么这个 SQL 的优化效果是很容易评估的，因为从 SQL 的执行开销到 SQL 的响应时间，都很容易获得。而如果针对一个系统优化的项目，优化效果的评估就不那么简单了。曾经有多个客户问我，优化后如何对结果进行评估呢？经常有人帮我们做了优化，而且优化后他们提交的评估数据都很漂亮，但是从业务部门得到的反馈和那些数据根本对不上。

确实评估优化效果最好的方法是最终用户的反馈，不过用户的反馈也不是绝对的，因为优化的目的有很多种，虽然说绝大多数系统优化的目的是提高系统的响应时间，不过我们不应该忘记，还有一些优化的目的可能是提高系统整体的吞吐能力。在这种情况下，响应时间并不是评估优化效果的最重要依据。因此针对不同的优化项目，优化评估的出发点是不同的。

虽然我们希望展现给最终客户的评估数据很漂亮，在选取评估数据时我们会尽可能选取漂亮一些的数据写在报告里，但是作为优化的实施者，必须清楚地认识到从事的优化工作的真实效果，我们需要做的是让用户真正认可我们的优化效果，而不是验收后客户又三番五次地找你的麻烦。与找到系统存在的问题相比，预估和评估优化的效果是更难的事情。

评估性能的提升首先需要从系统资源的情况来分析。优化实施后，任何一种资源使用情况必须在合理的范围内，如果优化后某个系统资源处于危险状态，则说明这次优化存在问题，必须继续优化，使资源使用更加合理。我们可以在优化前后采集操作系统 CPU、I/O 或内存的使用情况，进行对比。比如从下面的两个图表来看，7 月 19 日在业务高峰期间 9：10～10：10 和 15：10～16：10，系统比较平稳，CPU 空闲的比例超过了 30%，而 5 月 19 日优化前 CPU 的空闲比例小于 10%。

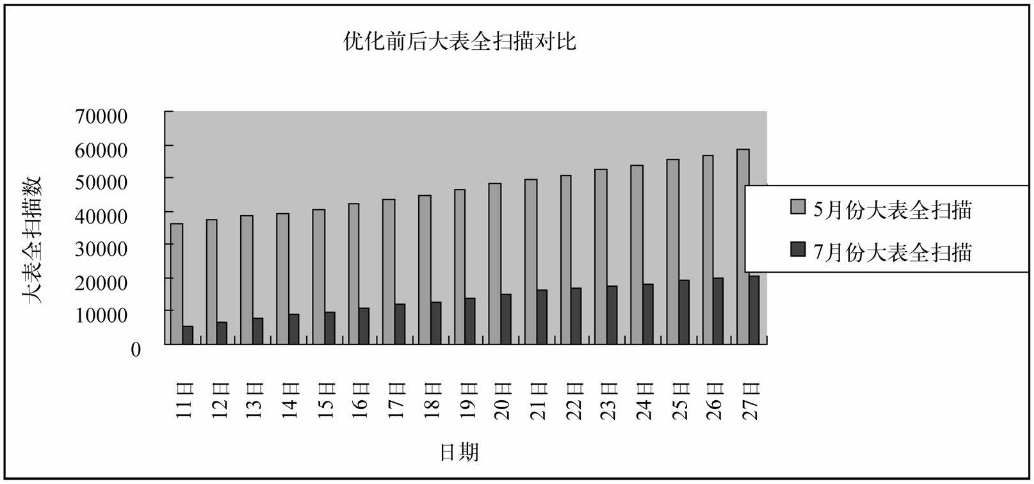




除了资源使用情况的比较外，从 STATSPACK（或者 AWR）报告中，可以提取出一些很有代表性的指标。这些指标具有一定的可比性。从事务响应时间、每秒事务数、每秒逻辑读、每秒物理读的数量等指标，都可以看出系统性能是否有所提升。由于每个应用系统在较短的时间区间内的特点（我们经常说的系统的节奏）是不会变的，所以多个 STATSPACK 报告的数据是可比的。不过为了能够获得更为准确的比对数据，需要选取两个业务相类似的报告进行比对。如何确定两份报告之间的业务相类似呢？如果可能的话，最好能够和开发部门或者业务部门的人一起来确定。比如，每个月中一套系统在不同时间段内跑的应用的特点可能略有不同，如很多运营商系统每个月底都会忙一些，月中时会闲一些，周末相对来说业务会少一些，平时业务更忙一些，等等。在每一天当中，系统也是有一定节奏的，比如每天早上 9：00～10：00 会比较忙，下午 3：00～4：00 会比较忙，下午 5：30～6：30 由于很多营业员扎账，可能导致系统负

载较大，等等。在掌握了系统的节奏后，我们很容易就能找出可比对的报告，并对系统的优化情况进行比对。因为优化不可能面面俱到，所以在不同的采样点的比对结果可能不同，有可能从每天业务高峰期间的比较来看，系统性能提升很大，但是如果比对扎账期间，性能提升可能不是特别明显，因为扎账时的应用可能没有进行过优化。实际上，最终写在报告里的评估值并不是最重要的。如果客户认可你的优化效果，那么报告里写什么样的值客户都可以认可，反过来客户如果不认可优化效果，你就是把报告写得再好客户也无法给你验收。

除了今天我们看到的那个比对结果外，很多其他方面的指标也可以进行比对，比如下图所示。



对于一个大表扫描十分严重的系统，我们如果要对应用进行优化，那么评估应用优化效果的一个重要指标是每秒大表全表扫描的数量。从上面的图中可以看出，优化后大表的全表扫描下降的幅度还是很明显的，说明应用优化起到了很好的效果。

优化小技巧 STATSPACK 报告分析之 undo 分析

STATSPACK 报告中关于 undo 分析的信息十分丰富。对于自动 undo 管理（AUM）来说，undo 需要调整的内容并不多，主要是 undo_retetion 和 undo 表空间的管理。从 STATSPACK 报告上，首先我们要分析 undo 段的总体情况：

```
        Trans Table       Pct   Undo Bytes
RBS No      Gets         Waits     Written        Wraps  Shrinks  Extends
------ -------------- ------- --------------- -------- -------- --------
     0       33,936.0    0.00               0        0        0        0
     1        2,200.0    0.00               0        0        0        0
     2       40,418.0    0.04      21,805,550        8        0        3
     3       33,175.0    0.02      20,182,728        3        0        0
     4       20,464.0    0.02       8,400,968        1        0        0
     5       19,714.0    0.03      11,173,194        4        1        0
     6       24,956.0    0.02       9,347,378        3        0        0
```

```
      7        81,818.0    0.02      196,187,192        55         0        47
      8        69,053.0    0.01      173,691,704        37         5        30
      9        16,216.0    0.02        6,262,656         7         0         0
     10        26,479.0    0.05       15,492,484         3         0         0
     28        20,994.0    0.02        7,631,770         8         0         0
     52        45,512.0    0.04       19,234,878         4         0         0
```

首先需要观察Pct Waits的比例。如果Pct Waits的数量超过1%，则说明undo段的数量过少。然后要观察一下Wraps、Shrinks、Extents的数量是否正常。如果这些值过高，就要分析undo是否存在问题，或者说是否有某个bug导致undo性能存在问题。通过下面的数据可以看出每个undo段的大小：

```
Rollback Segment Storage for DB: BOSSDB  Instance: boss1  Snaps: 28819 -28820
->Optimal Size should be larger than Avg Active

RBS No    Segment Size      Avg Active    Optimal Size    Maximum Size
------ --------------- --------------- --------------- ---------------
     0         385,024          64,543                         385,024
     1       8,511,488       2,862,784                     444,719,104
     2      36,823,040      41,857,688                     612,491,264
     3      33,677,312      17,271,927                   2,147,606,528
     4      41,017,344       3,312,216                   2,165,948,416
     5      42,065,920       9,242,569                   1,610,735,616
     6      37,871,616       8,214,914                   2,152,849,408
     7     276,946,944     225,785,851                   1,677,844,480
     8     251,781,120     214,080,651                   2,214,715,392
     9      30,531,584       2,285,847                   1,946,279,936
    10      35,774,464      75,155,736                   1,208,082,432
    28      34,725,888       1,209,811                   2,290,212,864
    52      45,211,648       4,962,510                   2,483,150,848
```

如果undo段的大小不均匀，存在较大的段，那么说明系统中的事务大小不一致，存在一些大型的事务。

Undo Segment Summary对于分析undo的情况十分重要，比如下面的数据：

```
Undo Segment Summary for DB: BOSSDB  Instance: boss1  Snaps: 28819 -28820
-> Undo segment block stats:
-> uS - unexpired Stolen,   uR - unexpired Released,   uU - unexpired reUsed
-> eS - expired   Stolen,   eR - expired   Released,   eU - expired   reUsed

Undo           Undo        Num  Max Qry     Max Tx Snapshot Out of uS/uR/uU/
 TS#         Blocks      Trans  Len (s)   Concurcy  Too Old  Space eS/eR/eU
---- -------------- ---------- -------- ---------- -------- ------ -------------
   1        271,537 ##########    2,845         32        1      0 0/0/0/0/0/0
          -------------------------------------------------------------
```

- ❑ Undo TS#：undo表空间号。
- ❑ Undo Blocks：undo块的数量。
- ❑ Num Trans：事务的数量。
- ❑ Max Qry Len(s)：undo表空间里最大查询的长度，这个值是确定undo_retention的重要参考。

- Max Tx Concurcy：最大的事务并发量。
- Snapshot Too Old：Snapshot Too Old 出现的次数。
- Out of Space：出现了 undo 表空间不足的次数。
- uS：unexpired Stolen，未预期的偷窃。Stolen 是从其他回滚段中偷取扩展的操作，unexpired Stolen 是偷取一个还未达到释放条件的扩展。只有当 undo 表空间不足时可能出现这种情况。
- uR：unexpired Released，未预期的释放。
- uU：unexpired reUsed，未预期的重用。

如果存在上述 3 种未预期的情况，可能说明 undo 表空间不足。下面的信息是以 10 分钟为单位的采样情况：

```
Undo Segment Stats for DB: BOSSDB  Instance: boss1  Snaps: 28819 -28820
-> ordered by Time desc

                     Undo        Num Max Qry   Max Tx  Snap   Out of uS/uR/uU/
End Time            Blocks     Trans Len (s)   Concy Too Old  Space eS/eR/eU
------------ ------------ -------- ------- --------- ------- ------ -------------
29-Jul 15:30       80,389 ########   1,216        26       0      0 0/0/0/0/0/0
29-Jul 15:20      133,397 ########   2,845        14       1      0 0/0/0/0/0/0
29-Jul 15:10       57,751 ########   2,367        32       0      0 0/0/0/0/0/0
          -------------------------------------------------------------
```

优化小技巧　STATSPACK 报告分析之锁分析

STATSPACK 报告中的锁分析较为简单，其中列出了所有锁等待的情况：

```
Enqueue activity for DB: BSS97  Instance: BSS97  Snaps: 40115 -40116
-> Enqueue stats gathered prior to 9i should not be compared with 9i data
-> ordered by Wait Time desc, Waits desc

                                                            Avg Wt         Wait
Eq     Requests    Succ Gets Failed Gets       Waits   Time (ms)     Time (s)
-- ------------ ------------ ----------- ----------- ------------- ------------
TX      373,140      373,136           5       5,200        110.03          572
WL            5            3           2           4     93,883.50          376
HW        2,428        2,084         344       1,356         77.25          105
US       30,128       30,128           0      21,440          1.74           37
TT       12,475       12,475           0       9,421          3.86           36
SQ       15,203       15,203           0       1,200         28.00           34
FB        2,088        2,088           0       1,027          2.62            3
CF        4,927        4,924           3         380          5.42            2
PS        3,428        2,092       1,336         356          3.44            1
TA           82           82           0          82          2.80            0
TS           32           32           0          16          2.75            0
MD           11           11           0          11          2.45            0
JQ           19            4          15           5          2.20            0
TM      655,079      654,984           0           2          3.50            0
          -------------------------------------------------------------
```

其中平均等待时间和总等待时间都是需要关注的。如果 enqueue 等待出现在 Top 事件中时就更加要关注这一节了。通过下面的脚本，可以了解系统中锁的详细情况：

```
set lines 200
set pagesize 66
spool locks.lis
break on Kill on sid on  username on terminal
column Kill heading 'Kill String' format a13
column res heading 'Resource Type' format 999
column id1 format 9999990
column id2 format 9999990
column locking heading 'Lock Held/Lock Requested' format a40
column lmode heading 'Lock Held' format a20
column request heading 'Lock Requested' format a20
column serial# format 99999
column username  format a10  heading "Username"
column terminal heading Term format a6
column tab format a30 heading "Table Name"
column owner format a9
column LAddr heading "ID1 - ID2" format a18
column Lockt heading "Lock Type" format a40
column command format a25
column sid format 990
rem     L.SID||','||S.SERIAL# Kill,

select
nvl(S.USERNAME,'Internal') username,
        L.SID,
        nvl(S.TERMINAL,'None') terminal,
        decode(command,
0,'None',decode(l.id2,0,U1.NAME||'.'||substr(T1.NAME,1,20),'None')) tab,
decode(command,
0,'BACKGROUND',
1,'Create Table',
2,'INSERT',
3,'SELECT',
4,'CREATE CLUSTER',
5,'ALTER CLUSTER',
6,'UPDATE',
7,'DELETE',
8,'DROP',
9,'CREATE INDEX',
10,'DROP INDEX',
11,'ALTER INDEX',
12,'DROP TABLE',
13,'---',
14,'---',
15,'ALTER TABLE',
16,'---',
17,'GRANT',
18,'REVOKE',
19,'CREATE SYNONYM',
20,'DROP SYNONYM',
```

```
21,'CREATE VIEW',
22,'DROP VIEW',
23,'---',
24,'---',
25,'---',
26,'LOCK TABLE',
27,'NO OPERATION',
28,'RENAME',
29,'COMMENT',
30,'AUDIT',
31,'NOAUDIT',
32,'CREATE EXTERNAL DATABASE',
33,'DROP EXTERNAL DATABASE',
34,'CREATE DATABASE',
35,'ALTER DATABASE',
36,'CREATE ROLLBACK SEGMENT',
37,'ALTER ROLLBACK SEGMENT',
38,'DROP ROLLBACK SEGMENT',
39,'CREATE TABLESPACE',
40,'ALTER TABLESPACE',
41,'DROP TABLESPACE',
42,'ALTER SESSION',
43,'ALTER USER',
44,'COMMIT',
45,'ROLLBACK',
46,'SAVEPOINT',
47,'PL/SQL EXECUTE',
48,'SET TRANSACTION',
49,'ALTER SYSTEM SWITCH LOG',
50,'EXPLAIN',
51,'CREATE USER',
52,'CREATE ROLE',
53,'DROP USER',
54,'DROP ROLE',
55,'SET ROLE',
56,'CREATE SCHEMA',
57,'CREATE CONTROL FILE',
58,'ALTER TRACING',
59,'CREATE TRIGGER',
60,'ALTER TRIGGER',
61,'DROP TRIGGER',
62,'ANALYZE TABLE',
63,'ANALYZE INDEX',
64,'ANALYZE CLUSTER',
65,'CREATE PROFILE',
66,'DROP PROFILE',
67,'ALTER PROFILE',
68,'DROP PROCEDURE',
70,'ALTER RESOURCE COST',
71,'CREATE SNAPSHOT LOG',
72,'ALTER SNAPSHOT LOG',
73,'DROP SNAPSHOT LOG',
74,'CREATE SNAPSHOT',
75,'ALTER SNAPSHOT',
```

```
76,'DROP SNAPSHOT',
84,'-',
85,'TRUNCATE TABLE',
86,'TRUNCATE CLUSTER',
87,'-',
88,'ALTER VIEW',
89,'-',
90,'-',
91,'CREATE FUNCTION',
92,'ALTER FUNCTION',
93,'DROP FUNCTION',
94,'CREATE PACKAGE',
95,'ALTER PACKAGE',
96,'DROP PACKAGE',
97,'CREATE PACKAGE BODY',
98,'ALTER PACKAGE BODY',
99,'DROP PACKAGE BODY',
command||' - ???') COMMAND,
        decode(L.LMODE,1,'No Lock',
                2,'Row Share',
                3,'Row Exclusive',
                4,'Share',
                5,'Share Row Exclusive',
                6,'Exclusive','NONE') lmode,
        decode(L.REQUEST,1,'No Lock',
                2,'Row Share',
                3,'Row Exclusive',
                4,'Share',
                5,'Share Row Exclusive',
                6,'Exclusive','NONE') request,
l.id1||'-'||l.id2 Laddr,
l.type||' - '||
decode(l.type,
'BL','Buffer hash table instance lock',
'CF','Cross-instance function invocation instance lock',
'CI','Control file schema global enqueue lock',
'CS','Control file schema global enqueue lock',
'DF','Data file instance lock',
'DM','Mount/startup db primary/secondary instance lock',
'DR','Distributed recovery process lock',
'DX','Distributed transaction entry lock',
'FI','SGA open-file information lock',
'FS','File set lock',
'IR','Instance recovery serialization global enqueue lock',
'IV','Library cache invalidation instance lock',
'MB','Master buffer hash table instance lock',
'MM','Mount definition gloabal enqueue lock',
'MR','Media recovery lock',
'RE','USE_ROW_ENQUEUE enforcement lock',
'RT','Redo thread global enqueue lock',
'RW','Row wait enqueue lock',
'SC','System commit number instance lock',
'SH','System commit number high water mark enqueue lock',
'SN','Sequence number instance lock',
```

```
'SQ','Sequence number enqueue lock',
'ST','Space transaction enqueue lock',
'SV','Sequence number value lock',
'TA','Generic enqueue lock',
'TD','DDL enqueue lock',
'TE','Extend-segment enqueue lock',
'TM','DML enqueue lock',
'TT','Temporary table enqueue lock',
'TX','Transaction enqueue lock',
'UL','User supplied lock',
'UN','User name lock',
'WL','Being-written redo log instance lock',
'WS','Write-atomic-log-switch global enqueue lock',
'TS',decode(l.id2,0,'Temporary segment enqueue lock (ID2=0)',
                    'New block allocation enqueue lock (ID2=1)'),
'LA','Library cache lock instance lock (A=namespace)',
'LB','Library cache lock instance lock (B=namespace)',
'LC','Library cache lock instance lock (C=namespace)',
'LD','Library cache lock instance lock (D=namespace)',
'LE','Library cache lock instance lock (E=namespace)',
'LF','Library cache lock instance lock (F=namespace)',
'LG','Library cache lock instance lock (G=namespace)',
'LH','Library cache lock instance lock (H=namespace)',
'LI','Library cache lock instance lock (I=namespace)',
'LJ','Library cache lock instance lock (J=namespace)',
'LK','Library cache lock instance lock (K=namespace)',
'LL','Library cache lock instance lock (L=namespace)',
'LM','Library cache lock instance lock (M=namespace)',
'LN','Library cache lock instance lock (N=namespace)',
'LO','Library cache lock instance lock (O=namespace)',
'LP','Library cache lock instance lock (P=namespace)',
'LS','Log start/log switch enqueue lock',
'PA','Library cache pin instance lock (A=namespace)',
'PB','Library cache pin instance lock (B=namespace)',
'PC','Library cache pin instance lock (C=namespace)',
'PD','Library cache pin instance lock (D=namespace)',
'PE','Library cache pin instance lock (E=namespace)',
'PF','Library cache pin instance lock (F=namespace)',
'PG','Library cache pin instance lock (G=namespace)',
'PH','Library cache pin instance lock (H=namespace)',
'PI','Library cache pin instance lock (I=namespace)',
'PJ','Library cache pin instance lock (J=namespace)',
'PL','Library cache pin instance lock (K=namespace)',
'PK','Library cache pin instance lock (L=namespace)',
'PM','Library cache pin instance lock (M=namespace)',
'PN','Library cache pin instance lock (N=namespace)',
'PO','Library cache pin instance lock (O=namespace)',
'PP','Library cache pin instance lock (P=namespace)',
'PQ','Library cache pin instance lock (Q=namespace)',
'PR','Library cache pin instance lock (R=namespace)',
'PS','Library cache pin instance lock (S=namespace)',
'PT','Library cache pin instance lock (T=namespace)',
'PU','Library cache pin instance lock (U=namespace)',
'PV','Library cache pin instance lock (V=namespace)',
```

```
'PW','Library cache pin instance lock (W=namespace)',
'PX','Library cache pin instance lock (X=namespace)',
'PY','Library cache pin instance lock (Y=namespace)',
'PZ','Library cache pin instance lock (Z=namespace)',
'QA','Row cache instance lock (A=cache)',
'QB','Row cache instance lock (B=cache)',
'QC','Row cache instance lock (C=cache)',
'QD','Row cache instance lock (D=cache)',
'QE','Row cache instance lock (E=cache)',
'QF','Row cache instance lock (F=cache)',
'QG','Row cache instance lock (G=cache)',
'QH','Row cache instance lock (H=cache)',
'QI','Row cache instance lock (I=cache)',
'QJ','Row cache instance lock (J=cache)',
'QL','Row cache instance lock (K=cache)',
'QK','Row cache instance lock (L=cache)',
'QM','Row cache instance lock (M=cache)',
'QN','Row cache instance lock (N=cache)',
'QO','Row cache instance lock (O=cache)',
'QP','Row cache instance lock (P=cache)',
'QQ','Row cache instance lock (Q=cache)',
'QR','Row cache instance lock (R=cache)',
'QS','Row cache instance lock (S=cache)',
'QT','Row cache instance lock (T=cache)',
'QU','Row cache instance lock (U=cache)',
'QV','Row cache instance lock (V=cache)',
'QW','Row cache instance lock (W=cache)',
'QX','Row cache instance lock (X=cache)',
'QY','Row cache instance lock (Y=cache)',
'QZ','Row cache instance lock (Z=cache)',
'TO','DDL operations on temporary tbl',
'DL','Direct Loader Index Creation',
'IN','Instance Number',
'IS','Instance State',
'KK','Redo Log "Kick" ',
'PI','Parallel Slaves',
'PR','Process Startup',
'PS','Parallel Slave Synchronization',
'SM','SMON',
'RO','Multiple object resue',
'????') Lockt
from    V$LOCK L,
        V$SESSION S,
        SYS.USER$ U1,
        SYS.OBJ$ T1
where   L.SID = S.SID
and     T1.OBJ#  = decode(L.ID2,0,L.ID1,1)
and     U1.USER# = T1.OWNER#
and     S.TYPE != 'BACKGROUND'
order by 1,2,5
/
```

7月21日 课堂风波

度假村在山里面，晚上十分安静，所以我睡得很好。早上迷迷糊糊地被烟味弄醒。肯定老于已经起来了，因为老于有起床一支烟的习惯。我睁开眼，一看表，是早上 6 点多。北方的天亮得真早，要在深圳我都以为是七八点钟了呢。

7 月份的沈阳早上气温适中，在山里面还是有一点清凉的感觉。湖边的空气很清新，沿湖的马路上已经有三三两两的人在湖边散步了。出门走了一段路，看到湖边不少人在钓鱼，用的都是很大的海竿。看到有个须发皆白的老者一边抽着烟一边盯着面前的十几根鱼竿，我走过去问道："大爷，有大鱼吗？"

大爷叼着烟说："有，你没看见我用的都是海竿？这湖里的鱼大的有几十斤。"

"真有这么大的鱼，还不早就让你们钓完了？"我表示不信。

"小伙子你还别不信，前年我们这里有个钓鱼的被鱼拉到湖中淹死了。他钓到一条大鱼不舍得松手，把鱼线缠到胳膊上了，几天后才从湖里浮上来，那叫惨呐!"看老爷子的意思不像是在开玩笑。

和老爷子聊了会天，又往前走了走，发现沿湖零零散散分布着不少度假村。散步回去，老于还在抽烟，我说老于别抽了，呆会你还要讲课呢，还是先吃饭吧。吃完饭我和老于一起到教室准备一下投影和扩音器材。今天上午除了北方公司领导和小齐做个开场白之外，就是老于介绍沈阳项目的总体情况，蓝本就是我们编写的《系统性能评估手册》和《系统优化方案》。讲《系统性能评估手册》时一切都很正常，沈阳项目的优化效果是让任何人都信服的。

不过在介绍《系统优化方案》时出现了一个小插曲。老于讲到修改 DB_FILES 参数时说，如果 DB_FILES 参数超出了控制文件的 MAXFILES 参数，就必须重建控制文件。当时下面就有人表示异议，说 DB_FILES 修改时不需要重建控制文件。那个小伙子是河北公司的老刘，也是一个从事维护工作四五年的老 DBA 了。老于和他探讨了几句，看他态度十分坚决，也有点拿不准主意了，就问了一句："老白，你觉得呢？"

我正坐在最后一排，准备着下午我的课程。虽然刚才也在听老刘和老于讨论，不过并没有深入去考虑这个问题。突然被问了一句，也有点拿不定主意。不过在我的印象里，如果文件数超过 MAXFILES，是需要重建控制文件的。因为控制文件被格式化后，文件槽位是固定的。于是我回答说："肯定要重建控制文件，否则没地方放文件信息。"长期以专家的身份和各种客户打交道，养成了我这种不容置疑的说话风格。听到我这么坚决的口气，老刘也就没有再坚持了。

中午吃饭时，老刘又找到我，希望我再确认一下，因为他还是认为自己的观点是对的。吃完饭回到房间，我马上上 Metalink 查了一下，发现我和老于确实错了。在文档 101020.1 里明确介绍了，从 Oracle 8.0 开始，控制文件就可以自动增长了，因此加大 `DB_FILES` 参数后，如果实际创建数据文件的个数超过了 `MAXFILES` 参数的限制，控制文件中的 `MAXFILES` 参数会自动增加，并且控制文件也会自动增长。这些年碰到类似的问题，我都会重建控制文件，从来没有怀疑过，看样子这个错误观点已经害了不少人了。我马上把正在午睡的老于叫了起来，说："今天我们两个确实错了，从 Oracle 8.0 开始控制文件可以自动增长了。你看怎么办？"

"那还能怎么办，下午主动认错吧。"老于飞快地看了一遍那个文档。说完这句话，老于掏出香烟，一声不吭地抽了起来。

"下午我的课，我来承认这个错误吧。你下午休息一下，不一定非要去教室了。"我觉得老于下午没课，专门为这件事去解释不太好。

下午一上课，我就把上午的错误向大家解释了一番，并且对上午提出疑问的老刘表示感谢。为了鼓励大家质疑和交流，我宣布赠送 10 个积分给老刘，以表彰他发现我们的问题。整个课堂里的气氛一下子热烈了起来，大家纷纷表态要多挑我的毛病，好获得积分。为了活跃课堂气氛，我们几个上课的人凑钱买了几个很不错的小礼品，并且在上午老于开课时就展示了这些小礼品，并宣布课程结束后，积分前三名的人会拿到这些小礼品。

晚上吃完晚饭，大家就自由活动了。由于我们住的度假村十分偏僻，没有汽车基本上没法出门，所以活动也只能在度假村里进行。好在度假村里的设施十分齐备，网球、保龄球、羽毛球、乒乓球、游泳、健身、桑拿、按摩、KTV、酒吧、烧烤一应俱全。组委会还给每个人发了 100 块钱的代金券，用于娱乐消费。我和老于两个人没别的爱好，洗了个澡就跑到房顶的烧烤吧喝酒了。今天上课出了一个纰漏，所以今天喝酒的话题总是免不了提到这个问题。现在知识更新这么快，而且 Oracle 技术那么繁杂，有个知识点没有更新也是难免的，我劝老于也别太当回事，不过话虽这么说，我自己也没办法释然。

老于又谈到了我们一个多月前谈到过的那个话题，是不是我们这批 DBA 已经老了，知识已经跟不上了。不过感慨归感慨，DBA 是一个职业，而不是某种知识，跟不上又能怎么样呢？到了我们这个年龄，学习新知识的能力肯定没办法和年轻人相比。不过我们现在的工作基本上是凭经验在做，而不是凭知识了，这是我们的优势。谈到这些，大家的心情又好了不少。

沈阳的烤鸡架确实不错，我们两个人每个人拿着一个烤鸡架，要了一打啤酒、几盘煮毛豆，喝得挺爽快。棋盘山的晚上比城里凉快得多，丝丝山风让人感到挺惬意的，坐在楼顶平台上喝酒简直就是一种享受。我们喝了一会，张工他们做完足底，也发现了这么个好地方，一起拥过来喝酒了。于是马上叫服务员搬了两箱啤酒过来，要了一些烤串，大家推杯换盏地喝了起来。明天还要继续上课，所以大家喝得都不多，不过一大群人吃着烤串喝着酒，聊聊风花雪月的事情，心情也好了不少。

大家喝到 1：00 左右才散了。明天上午是老于的 Oracle 数据库管理课程，下午是我的性能调优实战。老于回到酒店，打开电脑过了一遍明天的讲义，又抽了一支烟，才躺下。我觉得，今天

的事情在老于心里留下的阴影还是没有完全化解。说实在的，这件事我昨天晚上睡觉前也想了很久，做服务的人，积累下一些好名声并不容易，而让别人对你产生信任是十分关键的，举手投足都必须十分严谨，如果有人发现你在忽悠他，那么下回要让他相信你的话就难了。有了上午的事情，下午我讲课时，感觉互动的情况多了起来，质疑的声音也多了起来。这对于今后我们去他们那边实施优化项目十分不利，有什么办法能够震慑他们一下呢？算了，也别多想了，明天再说吧。

优化小技巧　平均事务响应时间的计算

书中经常会提到平均事务响应时间，实际上在优化项目中，平均事务响应时间是最为重要的指标之一，那么我们如何来计算平均事务响应时间呢？首先我们要理解，从 Oracle 数据库层面上计算的平均事务响应时间和客户应用系统的平均事务响应时间并不是一回事。从 Oracle 数据库层面上的平均事务响应时间是通过在一段时间内提交和回滚的数量为计数统计的总事务量。将一段时间内的总事务量除以时间就得出了平均事务数量，而平均事务响应时间就是基于平均事务数量的。而应用系统的事务概念可能和 Oracle 层面的事务完全无关。虽然 Oracle 层面的平均事务响应时间和应用层面的有较大的差别，但是平均事务响应时间这个指标值仍然在优化项目中具有很重要的参考意义。

那么平均事务响应时间该如何计算呢？通过 STATSPACK 报告或者 AWR 报告，可以很方便地计算出平均事务响应时间。首先，我们从 STATSPACK 报告中找到 Top Events（顶级事件）：

```
Top 5 Timed Events
~~~~~~~~~~~~~~~~~~                                                     % Total
Event                                            Waits    Time (s) Ela Time
-------------------------------------------- ------------ ----------- --------
db file scattered read                          1,572,376      23,056    34.83
db file sequential read                         1,719,609      18,768    28.35
buffer busy waits                                 621,320      10,200    15.41
CPU time                                                        8,487    12.82
PX Deq: Execute Reply                             243,188       2,183     3.30
          -------------------------------------------------------------
```

随便找一个等待事件，比如 db file scattered read，这个事件占整个系统的 34.85%。然后看下面的等待事件明细情况：

```
                                                                  Avg
                                                     Total Wait   wait    Waits
Event                               Waits   Timeouts   Time (s)   (ms)     /txn
---------------------------- ------------ ---------- ---------- ------ --------
db file scattered read          1,572,376          0     23,056     15     71.7
```

db file scattered read 这个事件，每个事务等待 71.7 次，每次等待 15ms，那么每个事务等待这个事件的时间是 71.7×15=1 075.5ms，而这个等待事件占整个系统等待时间的 34.85%，因此平均事务响应时间是：1075.5/0.3485=3 085ms。值得注意的是，由于在 STATSPACK 报告中，每次等待时间、平均事务等待次数等数值的取值是通过取整的，因此可能通过不同的等待事件计算出来的值会有些误差。更为精确的计算方法应该是平均事务响应时间为 23 056000/1 572 376×71.7/0.3485=3 016ms。

优化小技巧 STATSPACK 报告分析之闩锁分析

STATSPACK 报告是分析闩锁最好的工具之一。STATSPACK 报告中包含了大量闩锁的数据，从这些数据可以初步定位系统级的闩锁问题。要分析闩锁，首先要从 Load Profile 看起：

```
Instance Efficiency Percentages (Target 100%)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
            Buffer Nowait %:   99.93       Redo NoWait %:          100.00
            Buffer  Hit   %:   95.47    In-memory Sort %:           99.99
            Library Hit   %:   99.24        Soft Parse %:           97.68
         Execute to Parse %:   29.98         Latch Hit %:           99.75
Parse CPU to Parse Elapsd %:    2.12     % Non-Parse CPU:           96.75
```

这里的 Latch Hit 是在本报告期间总体的闩锁命中率，这个指标越接近 100%越好。如果这个值比较低（低于 99%一般来说闩锁的问题就相当严重了），那么在分析这份报告时，就要十分关注闩锁的问题了。虽然从命中率上我们看不出很明显的问题，不过在 Top 事件里：

```
Top 5 Timed Events
~~~~~~~~~~~~~~~~~~                                                     % Total
Event                                            Waits    Time (s) Ela Time
-------------------------------------------- ------------ ----------- --------
latch free                                      1,098,363     606,375    76.58
db file sequential read                        11,615,630      73,616     9.30
CPU time                                                       55,262     6.98
db file scattered read                          2,377,376      26,913     3.40
library cache pin                                   3,865      10,408     1.31
          -------------------------------------------------------------
```

可以看出 latch free 占整个等待事件的 76.58%。下面有关于闩锁的更为明细的信息：

```
Latch Activity for DB: HCSP  Instance: hcsp  Snaps: 111 -121
->"Get Requests", "Pct Get Miss" and "Avg Slps/Miss" are statistics for
  willing-to-wait latch get requests
->"NoWait Requests", "Pct NoWait Miss" are for no-wait latch get requests
->"Pct Misses" for both should be very close to 0.0

                                            Pct    Avg   Wait                 Pct
                              Get          Get   Slps   Time       NoWait NoWait
Latch                       Requests      Miss  /Miss    (s)     Requests   Miss
------------------------ -------------- ------ ------ ------ ------------ ------
Consistent RBA                  295,240    0.0             0            0
FAL request queue                    10    0.0             0            0
FIB s.o chain latch                 194    0.0             0            0
FOB s.o list latch               99,524    0.0    0.5      1            0
SQL memory manager latch              1    0.0             0        2,431    0.0
SQL memory manager worka        500,795    0.0    0.6     21            0
X$KSFQP                              49    0.0             0            0
active checkpoint queue          25,960    0.0             0            0
alert log latch                      23    0.0             0            0
archive control                     300    0.0             0            0
archive process latch               171    0.0             0            0
cache buffer handles          2,868,069    0.1    0.3    125            0
```

```
cache buffers chains       1,528,766,244    0.1    0.4 ######   65,503,162   23.5
cache buffers lru chain        1,667,682    0.6    0.2    782   69,088,014    3.8
cas latch                     52,531,336    0.1    0.3   3824            0
channel handle pool latc           6,653    0.0             0            0
channel operations paren          15,341    0.0             0            0
checkpoint queue latch         5,245,583    0.0    0.2      7      817,925    0.0
child cursor hash table        1,617,769    0.1    0.3     86            0
direct msg latch               1,411,172    0.0    0.0      0            0
dml lock allocation            1,714,958    0.2    0.5    304            0
done queue latch                 334,052    0.1    0.1      0      350,600    0.1
dummy allocation                   6,911    0.0             0            0
enqueue hash chains            4,946,060    0.1    0.5    752            0
enqueues                       4,910,923    0.2    0.5    815            0
event group latch                  3,325    0.0             0            0
global tx hash mapping           649,208    0.0    0.6      0            0
hash table column usage            1,169    0.3    0.5      5   32,482,472    5.0
```

在这个部分，我们重点观察 Pct Get Miss 和 Pct NoWait Miss 超过 1%的闩锁。比如在上面的数据中，cache buffers chains 和 cache buffers lru chain 的闩锁均存在问题，特别是 cache buffers chains 的 Pct NoWait Miss 达到 23.5%。下面我们进一步查看这两个闩锁的情况：

```
Latch Sleep breakdown for DB: HCSP  Instance: hcsp  Snaps: 111 -121
-> ordered by misses desc

                                      Get                                 Spin &
Latch Name                       Requests       Misses      Sleeps Sleeps 1->4
-------------------------- -------------- ----------- ----------- ------------
cache buffers chains        1,528,766,244   1,415,629     535,842 888231/518990/
                                                                  8373/35/0
row cache objects              84,271,679   1,375,515      70,469 1307405/65853/
                                                                  2170/87/0
row cache enqueue latch        80,394,155     866,473      10,018 856602/9724/
                                                                  147/0/0
library cache                  55,951,057     759,496     278,624 489396/261812/
                                                                  8068/220/0
shared pool                    19,702,554     249,585      43,609 207232/41159/
                                                                  1148/46/0
cas latch                      52,531,336      78,072      25,050 53208/24678/
                                                                  186/0/0
library cache pin              29,345,261      61,538      21,471 40280/21045/
                                                                  213/0/0
simulator lru latch             3,296,405      50,373      47,912 2854/47127/3
                                                                  91/1/0
library cache pin allocati     21,024,601      36,617      18,078 18655/17847/
                                                                  114/1/0
simulator hash latch           55,564,795      34,275       6,502 27824/6401/49/
                                                                  1/0
session allocation              7,403,893      30,725      14,038 16760/13892/
                                                                  73/0/0
session idle bit               54,312,472      21,733       5,213 16539/5175/19/
                                                                  0/0
multiblock read objects         7,538,869      19,500       6,013 13544/5899/
                                                                  57/0/0
```

```
cache buffers lru chain          1,667,682      10,703       2,660 8091/2564/
                                                                   48/0/0
enqueues                         4,910,923      10,523       5,096 5449/5052/22
                                                                   /0/0
```

上述数据是按照Misses的数量排序的。从这里可以看出cache buffers chains的Sleeps次数还是排在第一位，说明这个闩锁存在的问题最多。在一个小时内，Spin的次数居然高达888 231次，Sleep 1的次数达到518 990次，Sleep 2的次数为8 373，甚至出现了35次的Sleep三的情况。根据上面的情况来看，我们重点需要分析cache buffers chains的等待：

```
Latch Miss Sources for DB: HCSP  Instance: hcsp  Snaps: 111 -121
-> only latches with sleeps are shown
-> ordered by name, sleeps desc

                                                     NoWait              Waiter
Latch Name               Where                       Misses     Sleeps   Sleeps
------------------------ -------------------------- ------- ---------- --------
FOB s.o list latch       ksfd_allfob                      0          6        6
FOB s.o list latch       ksfdfol02                        0          1        1
SQL memory manager worka qesmmIRegisterWorkArea           0          5        2
SQL memory manager worka qesmmIUnRegisterWorkArea         0          2        1
cache buffer handles     kcbzfs                           0        253      257
cache buffer handles     kcbzgs                           0        246      242
cache buffers chains     kcbgtcr: kslbegin excl           0    227,599  414,454
cache buffers chains     kcbrls: kslbegin                 0    199,620   20,593
cache buffers chains     kcbgtcr: fast path               0     73,320   77,189
cache buffers chains     kcbzib: multi-block read:        0      7,494        0
cache buffers chains     kcbzgb: scan from tail. no       0      5,609        0
cache buffers chains     kcbzwb                           0      5,367   10,080
cache buffers chains     kcbchg: kslbegin: bufs not       0      4,870    2,647
cache buffers chains     kcbzib: finish free bufs         0      3,206    4,863
cache buffers chains     kcbchg: kslbegin: call CR        0      2,407    1,455
cache buffers chains     kcbgtcr: kslbegin shared         0      1,553    1,204
cache buffers chains     kcbgcur: kslbegin                0        884      910
cache buffers chains     kcbget: pin buffer               0        741    1,344
cache buffers chains     kcbgtcr                          0        702        1
cache buffers chains     kcbcge                           0        684       59
cache buffers chains     kcbget: exchange rls             0        667        2
cache buffers chains     kcbbic2                          0        267       24
cache buffers chains     kcbbxsv                          0        235      483
cache buffers chains     kcbnlc                           0        179      217
cache buffers chains     kcbget: exchange                 0        158       12
cache buffers chains     kcbzib: exchange rls             0         76        2
cache buffers chains     kcbbic1                          0         70      146
cache buffers chains     kcbnew_1                         0         43        2
cache buffers chains     kcbnew                           0         28       69
cache buffers chains     kcbibr                           0         27       43
cache buffers chains     kcbclc                           0          7        9
cache buffers chains     kcbesc: escalate                 0          2        0
cache buffers chains     kcbkzs                           0          2       10
cache buffers chains     kcbso1: set no access            0          2        0
cache buffers chains     kcbbwdb                          0          1        1
```

从上面等待源的分析可以看出这个闩锁等待产生的位置（在代码中的位置）。Oracle 内部的开发人员和GCS的技术人员可以很容易地定位等待的原因，比如排在第一位的是kcbgtcr: kslbegin excl,这是在做一致性读产生的。这部分分析比较专业，一般的 DBA 很难分析，不过可以由 Oracle 内部专门处理 bug 的技术人员来协助分析。

7月22日 世博园一日游和心想事成

昨天还算比较顺利，晚上又是烤鸡架和喝酒。因为今天没课了，所以昨晚大家喝得比较晚，还喝倒了几个。今天会务组组织去沈阳世博园参观。我没有睡懒觉的习惯，虽然昨天夜里2：00了才睡，今天早上7：00时还是自然醒了。老于昨天喝的有点多，今天早上还在沉睡。8：00才有早饭吃，所以决定到湖边散步。今天那个老头没来钓鱼，不禁又想起了他说的湖里有大鱼的事情。昨天晚饭时上了一条特别大的鲶鱼，一尺多的大盘子居然还装不下，厨师敲断了骨头把鱼盘过来居然尾巴和头连起来了，看样子连头带尾有接近一米长。

吃过早饭，大家坐了中巴去世博园。实际上，世博园就在棋盘山旁边，是以前的植物园改的。之前，我其实已经去过了，说实在的除了大并没有什么我很感兴趣的东西，花花草草的在北方可能算个稀奇物，在深圳是出门就看得到的。我订了今天下午3：00的航班回深圳。本来不想跟着大部队行动了，由于住在这个地方没车很不方便，想出去也就只能跟着大部队了。

车刚开出风景区大门，张工就从车尾走了过来，把电话交给我说："小孙说销账的数据库挂住了，什么操作都做不了，你帮着看看。"

听着张工这么一说，整车的人的目光都盯向了我。人要是运气来了是挡也挡不住，本来我一直想找个机会表现一下，为前天的纰漏弥补一下，这个机会真像是想睡觉时就有人递个枕头过来，想什么有什么啊。从目前销账数据库来看，能让所有操作都挂住的故障，可能性最大的就是归档目录满了。前几天也听说过磁带库最近总是有点问题，我还建议他清洗一下磁头。

为了让大家增加一些神秘感，我有意没有按照正常思路去处理这个问题。直接让小孙用`df -k`查一下文件系统的情况。小孙马上查了一下，发现/archive 目录满了。听到这个消息，我就知道这回赌对了。我装模作样地让小孙检查一下警告日志，果不其然，在警告日志里出现了归档无法完成的警告信息。于是我告诉车上的刘工，让他安排检查一下备份系统的情况，同时让小孙先把归档目录下的部分归档日志移到其他文件系统下。小孙那边的`mv`命令一发出，就发现系统恢复正常了。

我连忙把电话交给张工，说："故障排除了，剩下的问题要麻烦郭工检查一下备份系统是不是有问题。"

张工也有点惊讶："这就搞定了？你牛啊。"

我说："那当然，没这两手还敢给您老人家干活啊。"

从张工把电话交给我到我把电话交回给张工，整个处理过程不到5分钟。我感觉全车人看我

的眼光都有很大的变化。这个时候郭工也打完了电话，说确实是备份出问题了，昨天晚上的增量备份就没有成功。他已经安排现场值班的人去处理这个问题了，应该不会有什么大事。

从全车人的表情可以看出，我的目的完全达到了。和客户打交道，建立信任关系是十分关键的一件事情。我经常和公司的年轻人说，做服务三分靠技术，七分靠沟通。建立信任关系是沟通中十分重要的一点。今天我没有按照常规的方式，让小孙从警告日志和操作系统开始检查，就直奔主题，目的就是要让车里的人感受到，我们确实是专家，我们是值得信赖的。其实我也留有后手，如果目录没满，我会立即让小孙检查警告日志、操作系统、I/O 等情况。如果我没有赌对，其实也没什么影响，这种问题对于我来说基本上没有难度。这本来就是给我一次作秀的机会，关键是怎么表现得完美，今天这场秀做得完全成功。

在游园时，前天对控制文件重建提出异议的刘工和我说,他们的系统目前的问题不比沈阳少，这回北方公司安排的第二批优化项目里就有他们省，到时候希望我能去他们那里做实施。

我把刘工的原话告诉了小齐，小齐听了之后忍不住笑了：“今天你玩的这手太漂亮了，河北这边不太好打交道，而且系统问题也很多，我们也一直担心他们那边的项目不好干，既然他们有这个态度，倒是个好事情。”

中午和他们一起在世博园吃了点快餐，我就和大家分手了。他们游完世博园还准备晚上去中街刘老根大舞台看二人转。我今天就可以回家了，说实在的，这段时间一直在外面，每次回家时还是很兴奋的。

听说我要先走，郭工就说他也正好要回市里去处理磁带库的事情，他可以把我送到青年大街那边，省得我还要走老远去打车，于是我就坐了郭工的车子回市里。在路上，郭工问我今天怎么能这么快定位到这个问题，我说这就是经验了。排障是 DBA 的一项最基本的技能，大多数情况下，只要找到问题，解决问题都不难，每个 DBA 只要愿意下工夫去找，都没有问题。关键是每个人发现问题所需要的时间不同，这就取决于经验和技术能力。像今天的事情，从系统挂住到解决问题不到 10 分钟的时间，在运维工作中只要记录一个事件，而不用算是一次事故。如果换一个新手做，估计半个小时也不一定搞得好，而超过 30 分钟就必须记录一次事故了。能这么快定位问题，是十多年经验的积累，也只有对 Oracle 的基本原理搞得很清楚了，并且对所维护的系统十分了解，才能够第一时间去排查最可能出问题的部分。最近这两个月，我整天面对这几套系统做分析，已经对这些系统的情况了如指掌，所以才能做出如此快速的处理。面对一套新的系统，处理的时间可能要长一些了。

郭工听了我这番话，深有感触，说：“我们做甲方的，想达到这个境界基本上没什么希望了。杂事太多，根本没有时间认真研究一下技术，研究一下系统。”确实也是，我曾经和好多甲方的工程师合作过，他们以前都是 Oracle 方面的高手，而随着自己在企业里越来越资深，从事的管理工作越来越多，时间长了，技术方面就生疏了，甚至很多人只能停留在理论和概念上了。DBA 这个职业，大多数情况下只有在第一线工作的人才有动力去继续学习，一旦离开生产第一线，技术退化就只是时间的问题了。

今天飞机还算准点，刚通过安检，就听到了航班登机的预告。今天在飞机上，躺在舒适的头

等舱里，闭目思考，确实十分惬意。这个项目，从 5 月初的迷茫到 6 月初的无奈，最后到今天的皆大欢喜，真像一出戏一样。从这个项目里我又学到了很多东西，从老于身上我体会到了什么叫责任，从老熊身上我体会到了什么叫执着，从老方身上我体会到了什么叫决断，从老肖身上我体会到了什么叫坚持。每个项目结束后，认真回顾和总结一下是我的习惯。这种回顾一般都是非正式的，不过我会花上一些时间来冥想，每次冥想都会给我带来收获。

优化小技巧 STATSPACK 报告分析之 Top 对象

STATSPACK 的采样可以分为几种级别。默认情况下，Snap 是 5 级的。我们也可以通过 exec statspack.Snap(i_snap_level=>6)来获得 6 级的采样。使用 6 级或者 6 级以上的采样生成的报告中有一个十分有用的章节，就是 Top 段。比如：

```
Top 10 Logical Reads per Segment for DB: TBCS  Instance: tbcs1  Snaps: 37838 -37
-> End Segment Logical Reads Threshold:     10000

                                           Subobject  Obj.       Logical
Owner      Tablespace Object Name          Name       Type         Reads  %Total
---------- ---------- -------------------- ---------- ----- ------------ -------
TBCS       INX        PK_PRIVILEGE_ORG                INDEX  396,406,768   28.48
TBCS       DATA       PARAMETER_VALUE                 TABLE  260,254,096   18.70
TBCS       DATA       ACCAGENTUNIT                    TABLE   24,922,480    1.79
TBCS       DATA       DICT_ITEM                       TABLE   17,116,976    1.23
TBCS       DATA       PRODUCT_PRIVILEGE               TABLE   16,738,672    1.20
IBOSS      DATA       TRANSLOG                        TABLE   12,402,160     .89
TBCS       DATA       TX_LOGIN_MSG                    TABLE   11,570,976     .83
TBCS       INX311     PK_SSC               311_200505 INDEX   10,679,200     .77
TBCS       IDX310     PK_SSC               310_200505 INDEX   10,673,040     .77
TBCS       IDX314     PK_SUBSCRIBER        CRIBER_314 INDEX   10,261,408     .74
          -------------------------------------------------------------


Top 10 Physical Reads per Segment for DB: TBCS  Instance: tbcs1  Snaps: 37838 -3
-> End Segment Physical Reads Threshold:    1000

                                           Subobject  Obj.      Physical
Owner      Tablespace Object Name          Name       Type         Reads  %Total
---------- ---------- -------------------- ---------- ----- ------------ -------
MISC       DATA       TRANSLOG             LOG_200506 TABLE      159,619    2.59
TBCS       IDX310     PK_SOCIAL_ENTITY2    ENTITY_310 INDEX      151,215    2.46
TBCS       DATA310    SUBSCRIBER           CRIBER_310 TABLE      147,184    2.39
TBCS       DATA312    SUBSCRIBER           CRIBER_312 TABLE      142,406    2.31
MISC       DATA       TRANSLOG             LOG_200508 TABLE      116,599    1.89
MISC       DATA       TRANSLOG             LOG_200507 TABLE      115,775    1.88
TBCS       DATA315    SUBSCRIBER           CRIBER_315 TABLE      114,218    1.86
TBCS       INX315     PK_SOCIAL_ENTITY2    ENTITY_315 INDEX      104,659    1.70
TBCS       INX312     PK_SOCIAL_ENTITY2    ENTITY_312 INDEX      100,699    1.64
IBOSS      DATA       TRANSLOG                        TABLE       91,783    1.49
          -------------------------------------------------------------
Top 10 Buf. Busy Waits per Segment for DB: TBCS  Instance: tbcs1  Snaps: 37838 -
-> End Segment Buffer Busy Waits Threshold:     100
```

```
                                                                 Buffer
                                           Subobject  Obj.         Busy
Owner      Tablespace Object Name          Name       Type        Waits  %Total
---------- ---------- -------------------- ---------- ----- ------------ -------
MISC       DATA       TRANSLOG             LOG_200508 TABLE       13,588   16.44
MISC       DATA       TRANSLOG             LOG_200507 TABLE       11,386   13.78
TBCS       IDX310     PK_SOCIAL_ENTITY2    ENTITY_310 INDEX        7,626    9.23
MISC       DATA       TRANSLOG             LOG_200506 TABLE        6,942    8.40
TBCS       DATA       PK_AGENTCHARGE_CHECK ECK_200508 INDEX        5,962    7.21
TBCS       DATA       AGENTCHARGE_CHECK    ECK_200508 TABLE        4,483    5.42
TBCS       INX312     PK_SOCIAL_ENTITY2    ENTITY_312 INDEX        3,691    4.47
IBOSS      INX        PK_STATCHGNTFYREQ    REQ_200508 INDEX        2,892    3.50
IBOSS      INX        INX_STATCHGNTFYREQ   REQ_200508 INDEX        1,799    2.18
TBCS       INX315     PK_SOCIAL_ENTITY2    ENTITY_315 INDEX        1,615    1.95
          -------------------------------------------------------------


Top 10 Row Lock Waits per Segment for DB: TBCS  Instance: tbcs1  Snaps: 37838 -3
-> End Segment Row Lock Waits Threshold:     100

                                                                    Row
                                           Subobject  Obj.         Lock
Owner      Tablespace Object Name          Name       Type        Waits  %Total
---------- ---------- -------------------- ---------- ----- ------------ -------
TBCS       DATA312    PK_CYCLEINV_BILLITEM 312_200508 INDEX          274    6.39
TBCS       DATA312    OPERATOR_FORMNUM     ORMNUM_312 TABLE          180    4.19
IBOSS      INX        PK_STATCHGNTFYREQ    REQ_200508 INDEX          150    3.50
TBCS       DATA310    PK_CYCLEINV_BILLITEM 310_200508 INDEX          145    3.38
TBCS       DATA315    PK_CYCLEINV_BILLITEM 315_200508 INDEX          103    2.40
TBCS       DATA313    PK_CYCLEINV_BILLITEM 313_200508 INDEX           94    2.19
MISC       INX        PK_STATCHGNTFYREQ    REQ_200508 INDEX           93    2.17
TBCS       DATA312    PK_SWC_REDUNDANT     312_200508 INDEX           67    1.56
TBCS       DATA311    PK_CYCLEINV_BILLITEM 314_200508 INDEX           57    1.33
TBCS       DATA312    IDX_EXECTIME_SWITCHC 312_200508 INDEX           53    1.24
          -------------------------------------------------------------


Top 10 ITL Waits per Segment for DB: TBCS  Instance: tbcs1  Snaps: 37838 -37882
-> End Segment ITL Waits Threshold:     100

                                           Subobject  Obj.          ITL
Owner      Tablespace Object Name          Name       Type        Waits  %Total
---------- ---------- -------------------- ---------- ----- ------------ -------
TBCS       DATA315    PK_CYCLEINV_BILLITEM CLEINVOICE INDEX           12   10.43
TBCS       DATA312    PK_CYCLEINV_BILLITEM CLEINVOICE INDEX            8    6.96
TBCS       DATA315    IDX_STATQ_CHARGE_FIN STATQ_CHAR INDEX            7    6.09
TBCS       DATA315    PK_SWC_REDUNDANT     SWITCH_COM INDEX            5    4.35
TBCS       DATA312    PK_FRAUD_CONTROL_ACC AUD_CONTRO INDEX            4    3.48
TBCS       DATA311    PK_CYCLEINV_BILLITEM CLEINVOICE INDEX            4    3.48
TBCS       DATA312    IDX_FRAUDLOG_CTRLTIM FRAUD_CONT INDEX            4    3.48
IBOSS      INX        PK_STATCHGNTFYREQ    STATCHGNTF INDEX            3    2.61
TBCS       DATA310    PK_SWC_REDUNDANT     SWITCH_COM INDEX            3    2.61
TBCS       DATA313    PK_RECEPTION         RECEPTION_ INDEX            3    2.61
```

```
          -------------------------------------------------------------
Top 10 CR Blocks Served (RAC) per Segment for DB: TBCS  Instance: tbcs1  Snaps:
-> End Global Cache CR Blocks Served Threshold:    1000

                                                                    CR
                                         Subobject  Obj.        Blocks
Owner      Tablespace Object Name         Name       Type        Served  %Total
---------- ---------- -------------------- ---------- ----- ------------ -------
TBCS       INX312     INDEX_SMJOBNOTIFY_TR _312_LESSI INDEX       41,922   10.69
TBCS       IDX310     INDEX_SMJOBNOTIFY_TR _310_LESSI INDEX       39,374   10.04
TBCS       INX315     INDEX_SMJOBNOTIFY_TR _315_LESSI INDEX       36,965    9.43
IBOSS      DATA       STATCHGNTFYREQ       REQ_200508 TABLE       32,272    8.23
TBCS       DATA       IDX_SMP_COMM_RESULTC OMM_200508 INDEX       27,556    7.03
TBCS       IDX314     INDEX_SMJOBNOTIFY_TR _314_LESSI INDEX       21,635    5.52
TBCS       IDX313     INDEX_SMJOBNOTIFY_TR _313_LESSI INDEX       20,718    5.28
TBCS       DATA       SMP_COMM             OMM_200508 TABLE       15,055    3.84
IBOSS      INX        PK_STATCHGNTFYREQ    REQ_200508 INDEX       13,798    3.52
TBCS       DATA312    SMJOBNOTIFY          _312_LESSI TABLE        7,882    2.01
          -------------------------------------------------------------


Top 10 CU Blocks Served (RAC) per Segment for DB: TBCS  Instance: tbcs1  Snaps:
-> End Global Cache CU Blocks Served Threshold:    1000

                                                                    CU
                                         Subobject  Obj.        Blocks
Owner      Tablespace Object Name         Name       Type        Served  %Total
---------- ---------- -------------------- ---------- ----- ------------ -------
IBOSS      DATA       STATCHGNTFYREQ       REQ_200508 TABLE       32,454    8.24
IBOSS      INX        INX_STATCHGNTFYREQ   REQ_200508 INDEX       23,108    5.87
IBOSS      INX        PK_STATCHGNTFYREQ    REQ_200508 INDEX       21,119    5.36
MISC       INX        PK_STATCHGNTFYREQ    REQ_200508 INDEX       17,251    4.38
PERFSTAT   USERS      STATS$SQL_PLAN_USAGE            INDEX        9,932    2.52
TBCS       DATA       PRIVILEGE_ORG                   TABLE        9,155    2.33
TBCS       INX312     INDEX_SMJOBNOTIFY_TR _312_LESSI INDEX        9,046    2.30
TBCS       IDX310     INDEX_SMJOBNOTIFY_TR _310_LESSI INDEX        8,963    2.28
SYS        SYSTEM     SEQ$                            TABLE        8,400    2.13
TBCS       INX315     INDEX_SMJOBNOTIFY_TR _315_LESSI INDEX        7,859    2.00
          -------------------------------------------------------------
```

Top段是根据不同的排序方式列出了开销最大的几个对象（默认情况是5个，可以通过修改sprepins.sql中的参数来修改列出的段的数量）：

```
define top_n_events = 5;     --顶级事件的显示数量，默认 5 个
define top_n_sql = 65;       --顶级 SQL 的显示数量，默认 65 个
define top_n_segstat = 5;    --顶级对象的显示数量，默认 5 个
define num_rows_per_hash=5; --每个 SQL 显示的最多行数，默认 5 行
```

Top段有助于我们分析系统。比如我们要查找缓冲区获取较多的段，拿本例来看吧：

```
TBCS       INX        PK_PRIVILEGE_ORG        INDEX   396,406,768   28.48
TBCS       DATA       PARAMETER_VALUE         TABLE   260,254,096   18.70
```

这一张表和一个索引占了整个系统缓冲区获取的47%以上。如果要进行优化，针对这两个对象进

行优化是比较有效的。如果我们要查找 INI_TRANS 参数设置不合理的对象，可以从 ITL 等待最多的对象找起：

```
Top 10 ITL Waits per Segment for DB: TBCS  Instance: tbcs1  Snaps: 37838 -37882
-> End Segment ITL Waits Threshold:     100

                                           Subobject  Obj.            ITL
Owner      Tablespace Object Name          Name       Type          Waits  %Total
---------- ---------- -------------------- ---------- ----- ------------ -------
TBCS       DATA315    PK_CYCLEINV_BILLITEM CLEINVOICE INDEX           12   10.43
TBCS       DATA312    PK_CYCLEINV_BILLITEM CLEINVOICE INDEX            8    6.96
TBCS       DATA315    IDX_STATQ_CHARGE_FIN STATQ_CHAR INDEX            7    6.09
```

这几个索引在 ITL 等待上比较多，因此我们可以考虑加大这几个索引的 INI_TRANS 参数。

优化小技巧 STATSPACK 报告分析之 Top SQL

查找 Top SQL(顶级 SQL)的工作是优化工作中经常遇到的。STATSPACK 报告是查找 Top SQL 十分好的工具。根据不同的目的，查找 Top SQL 有不同的过滤条件。作为系统整体优化来说，我们需要找出对系统影响最大的 SQL，查找的过滤条件是根据总的缓冲区获取或者读来排序。而如果我们的目的是查找性能存在问题的 SQL，那么我们的过滤条件可能是单条 SQL 的开销，而不是在某个时间段内的总开销。因此，两种方法找出来的 SQL 可能不同。如果是做系统优化，老白喜欢通过 STATSPACK 报告来查找 Top SQL。由于 STATSPACK 报告中每个 SQL 默认显示 5 行，而对于绝大多数的 Top SQL 来说，大多数超过 5 行，所以在生成报告前，最好将 num_rows_per_hash 参数加大为 10 或者更大。下面是一个例子：

```
SQL ordered by Gets for DB: BSS  Instance: BSS  Snaps: 4243 -4244
-> End Buffer Gets Threshold:   10000
-> Note that resources reported for PL/SQL includes the resources used by
   all SQL statements called within the PL/SQL code.  As individual SQL
   statements are also reported, it is possible and valid for the summed
   total % to exceed 100

                                                     CPU      Elapsd
  Buffer Gets    Executions  Gets per Exec  %Total Time (s)  Time (s) Hash Value
--------------- ------------ -------------- ------ -------- --------- ----------
        581,930           52       11,191.0    7.1    11.47     36.98 3229205788
Module: JDBC Thin Client
select * from (select star.*,rownum as numrow from ( select nvl(
 a.return_flag, ' ' ),nvl(a.service_grade||' '||f.SERVICE_GRADE_
NAME, ' ' ),  decode(nvl(a.cust_grade,' '),'1','一般客户','2','
重要客户','3','至关重要客户','其他') cust_grade_aa,  nvl( a.shee
t_no, ' ' ),      nvl( a.serv_no, ' ' ),       nvl( a.new_branch

        512,733          127        4,037.3    6.3    43.22    121.90 3717445708
Module: JDBC Thin Client
select order_no ord_no,nvl(num,'') num,decode(action_type,1,'Cre
ate',2,'Update',3,'Delete') action_type,id_flag,action_type acti
on_type_idx  from tif_bill_order_ctr where proc_flag='0'  order
```

```
by TO_NUMBER(trim(id_flag)),TO_NUMBER(trim(action_type_idx))

        449,714           63        7,138.3    5.5     5.22     41.63  922404172
Module: JDBC Thin Client
 select a.sell_type,a.cons_id ,b.CONS_NAME from tcm_user a ,TAM_
CONSIGNEE b where a.user_seq >0 and   a.STATE = '1' and   a.cons
_id = b.CONS_EXTER_CODE and   b.state ='1' and   a.finish_flag =
'0' and   a.ord_no =:1 union  select a.sell_type,a.cons_id ,b.cm
_id || b.name as CONS_NAME from tcm_user a ,TWM_CUST_MANAGER b w
```

我们看到散列值是 3229205788 的 SQL 的缓冲区获取占整个系统的 7.1%，对这个 SQL 进行优化可以提升系统性能。在以往的优化项目中，我们经常能够看到占系统总体缓冲区获取或者读超过 30%的 SQL，这种 SQL 就是我们优化的重点。通过 sprepsql.sql 脚本，可以查看更为详细的信息：

```
STATSPACK SQL report for Hash Value: 3229205788  Module: JDBC Thin Client

DB Name         DB Id    Instance     Inst Num Release     Cluster Host
------------ ----------- ------------ --------- ----------- ------- ------------
BSS           36751116 BSS                  1 9.2.0.6.0   NO      LN-CRM-DB-A

 Start Id     Start Time         End Id      End Time       Duration(mins)
--------- ------------------- --------- ------------------- --------------
     4243 30-5月 -06 09:22:33      4244 30-5月 -06 09:28:33           6.00

SQL Statistics
~~~~~~~~~~~~~~
-> CPU and Elapsed Time are in seconds (s) for Statement Total and in
   milliseconds (ms) for Per Execute
                                                       % Snap
                     Statement Total      Per Execute   Total
                     ---------------  ---------------  ------
        Buffer Gets:         581,930         11,191.0    7.10
         Disk Reads:           1,641             31.6     .12
     Rows processed:             363              7.0
     CPU Time(s/ms):              11            220.6
 Elapsed Time(s/ms):              37            711.1
              Sorts:              52              1.0
        Parse Calls:              52              1.0
      Invalidations:               0
      Version count:               1
    Sharable Mem(K):             135
         Executions:              52

SQL Text
~~~~~~~~
SELECT * FROM (SELECT star.*,rownum as numrow FROM ( SELECT nvl( a.return_flag, ' ' ),
nvl(a.service_grade||' '||f.SERVICE_GRADE_NAME, ' ' ),
decode(nvl(a.cust_grade,' '),'1','一般客户','2','重要客户','3','至关重要客户','其他')
cust_grade_aa,
nvl( a.sheet_no, ' ' ),     nvl( a.serv_no, ' ' ),      nvl( a.new_branch_no, ' ' ),
nvl( a.new_num, ' ' ),      nvl( a.install_flag, '' ),  nvl( a.old_branch_no, ' ' ),
nvl( a.old_num, ' ' ),
nvl( a.sheet_state, ' ' ),   nvl( a.ord_no, ' ' ) ord_no, nvl( a.ask_no, ' ' ),
```

```
nvl( a.spec_flag, ' ' ),     nvl(to_char( a.besp_date, 'yyyy-mm-dd hh24:mi:ss' ), '
' ),
nvl( a.sheet_type, ' ' ),  a.bureau_no,  nvl( a.notes, ' ' ),  nvl( a.send_site_no,
' ' ),
nvl( a.send_staff_no, ' ' ), nvl( to_char( a.send_sheet_date, 'yyyy-mm-dd hh24:mi:ss' ),
' ' ),
nvl( to_char( a.crt_sheet_date,  'yyyy-mm-dd hh24:mi:ss' ), ' ' ),  ' ',
 nvl( a.user_seq, 0 ),  nvl( a.prev_bureau_no, ' ' ),  nvl( a.prev_ord_no, ' ' ),
 nvl( a.prev_user_seq, 0 ),  nvl( e.user_name, ' '),  nvl( e.addr_detail, ' ')
addr_detail
 from tom_sheet_manage a, tom_exec_order d,tcm_user e,tvlcm_service_grade f  where
a.sheet_type like :1||'%'  and a.seq = 0  and a.sheet_state = '1'   and a.msg_flag =
'0'
and a.return_flag <> '2'  and a.return_flag <> :2  and exists
( SELECT /*+ hash_sj */ 1  FROM tvlom_site_to_info b  WHERE a.dept_code=b.dept_code
and b.site_no =:3
and b.site_purview='1'  and b.bureau_no=a.bureau_no  and b.seq = 0 )
and a.ord_no = d.ord_no  AND d.USER_ID = e.USER_ID  and d.user_seq = e.user_seq  and
a.service_grade = f.service_grade  and
f.STATE='1'  and d.seq = (select max(f.seq) from tom_exec_order f
where d.ord_no = f.ord_no) order by ord_no,addr_detail desc) star )
where numrow<=(:4*:5) and numrow>((:6-1)*:7)

Plans in shared pool between Begin and End Snap Ids
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Shows the Execution Plans found in the shared pool between the begin and end
snapshots specified.  The values for Rows, Bytes and Cost shown below are those
which existed at the time the first-ever snapshot captured this plan - these
values often change over time, and so may not be indicative of current values
-> Rows indicates Cardinality, PHV is Plan Hash Value
-> ordered by Plan Hash Value

--------------------------------------------------------------------------------
| Operation                         | PHV/Object Name      |  Rows | Bytes|   Cost |
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------

End of Report
```

可以看到，在这个报告里，这个 SQL 的完整的 SQL Text 被显示了出来，而且其开销也清晰地列了出来。可以看到，报告最后的执行计划部分是空的，这是因为该报告的采样级别是默认的 5 级。如果你的 Snap Level 是 6 或者更高的，那么执行计划也会详细显示。在做 SQL 优化时，我们经常会要求客户采集 Snap Level 为 6 的采样点，这样便于我们进行分析。对于 Oracle 10g 的数据库，在 AWR 报告中，默认的采样就包含了 SQL 的执行计划，这对于我们分析 SQL 是很有帮助的。比如下面的案例就是从一份 6 级的 STATSPACK 报告上取下来的：

```
STATSPACK SQL report for Hash Value: 1144597049  Module: JDBC Thin Client

DB Name         DB Id    Instance     Inst Num Release     Cluster Host
------------ ----------- ------------ -------- ----------- ------- ------------
BSS             38876318 BSS                 1 9.2.0.5.0   NO      HB-CRM-DB-A
```

```
 Start Id     Start Time         End Id      End Time       Duration(mins)
--------- ------------------- --------- ------------------- --------------
    15870 16-12 月-06 08:00:02     15871 16-12 月-06 09:00:04          60.03

SQL Statistics
~~~~~~~~~~~~~~
-> CPU and Elapsed Time are in seconds (s) for Statement Total and in
   milliseconds (ms) for Per Execute
                                                       % Snap
                     Statement Total      Per Execute   Total
                     ---------------  ---------------  ------
        Buffer Gets:      25,722,170            388.4   20.31
         Disk Reads:               1              0.0     .00
     Rows processed:           1,957              0.0
     CPU Time(s/ms):             276              4.2
 Elapsed Time(s/ms):             373              5.6
              Sorts:             671               .0
        Parse Calls:          66,253              1.0
      Invalidations:               0
      Version count:               4
    Sharable Mem(K):              39
         Executions:          66,221

SQL Text
~~~~~~~~
SELECT BUREAU_NO,  ORD_NO,          ACTION_NO,       DISPATCH_NO
,     nvl(SITE_NO,' '),         nvl(STAFF_NO,' '),         MSG_N
O           FROM TOM_MSG A  WHERE MSG_FLAG = '0'  AND EXISTS (SE
LECT 1 FROM TOM_EXEC_ORDER B  WHERE A.ORD_NO = B.ORD_NO  AND ASK
_NO LIKE  :1 )  order by msg_occur_date

All Optimizer Plan(s) for this Hash Value
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Shows all known Optimizer Plans for this Hash value, and the Snap Id's they
were first found in the shared pool
-> ordered by Snap Id

    Plan
 Hash Value  Snap Id       Cost Optimizer
------------ -------- ---------- --------------------
  3133398659    14880        17  CHOOSE
  1727509485    15811       567  CHOOSE
  3133398659    15863        48  CHOOSE
  3133398659    16009        80  CHOOSE
  3133398659    16013       253  CHOOSE
  3133398659    16114        23  CHOOSE
  2163395318    16145       126  CHOOSE

Plans in shared pool between Begin and End Snap Ids
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Shows the Execution Plans found in the shared pool between the begin and end
snapshots specified.  The values for Rows, Bytes and Cost shown below are those
which existed at the time the first-ever snapshot captured this plan - these
values often change over time, and so may not be indicative of current values
```

```
-> Rows indicates Cardinality, PHV is Plan Hash Value
-> ordered by Plan Hash Value

--------------------------------------------------------------------------------
| Operation                        | PHV/Object Name     |  Rows | Bytes|   Cost |
--------------------------------------------------------------------------------
|SELECT STATEMENT                  |----- 3133398659 ----|       |      |     17 |
|SORT ORDER BY                     |                     |     1 |   95 |     17 |
| NESTED LOOPS SEMI                |                     |     1 |   95 |      8 |
|  TABLE ACCESS BY INDEX ROWID     |TOM_MSG              |     1 |   63 |      6 |
|   INDEX RANGE SCAN               |IND_MSG_FLAG         |     1 |      |      5 |
|  TABLE ACCESS BY INDEX ROWID     |TOM_EXEC_ORDER       |    1K|   45K|      2 |
|   INDEX RANGE SCAN               |FK_TOM_EXEC_ORDER    |     1 |      |      1 |
--------------------------------------------------------------------------------

End of Report
```

7月23日 漫长的一夜

飞机到深圳时已经是晚上6点多了，我拖着疲惫的身体，正准备回家，突然一个电话打了进来，是我们公司的一名销售人员。他问我能不能马上去趟广州，有个客户在Sun公司做一个应用软件测试，在测试过程中出现了严重的性能问题，已经搞了4天了，还是没有一点进展，明天上班前如果不能搞定，就要出大问题了。

刚经过了4个多小时的旅程，人很疲惫，而且快一个星期没回家了，一听到这个消息，刚开始的反应是能不能推掉。不过听他说，客户那边确实比较紧急，而且客户是国内最大的运营商之一，也是一个拓展客户的好机会，所以最后我还是答应了。在机场直接叫了辆出租车，对司机说去广州。司机楞了一下说，“哥们，你晕了吧，去广州还是深圳？”听我确定是广州，司机马上乐开了花，你是讲价还是打表，如果打表可能要400元。我说打表吧，反正有人报销。

在出租车上，我和广州的联系人联络了一下，了解一下那边的情况，以便于到现场时可以更快处理。客户那边在测试一个应用，服务器是一台Sun FIRE 4900，带一个盘阵，16块盘，每8块组成一个RAID 0，然后组成RAID 0+1，只划了一个LUN，安装的是Solaris 10，数据库版本是Oracle 10.2.0.3。在测试时，他们发现某个应用模块第一次执行48s，第二次执行1分半，每次执行都会比前一次慢几十秒，如果会话重新连接，又会恢复为48s，然后逐渐衰减。后来他们在一台安装了 Windows 虚拟机的笔记本电脑上进行测试，发现在微机上运行十分稳定，都是平均25s左右。这里就有两个问题了，一是为什么这个应用在Sun FIRE 4900小型机上跑，比性能差很多的笔记本电脑还要慢那么多；二是为什么这个应用会越执行越慢。在这之前，他们已经做了4天的优化，大量的DBA和Sun的技术人员对系统做了各种优化尝试，均以失败告终。从现象上看，今天我可能遇到了一个十分棘手的问题。

晚上7点多，我到达了位于大都会广场的Sun广州办事处，测试环境就搭建在那里。由于明天下午下班前，这个测试必须完成，否则本次测试就宣告失败了。解决问题后还需要在测试环境中装载大量的测试数据，因此客户给我进行优化的最后期限是明天早上8：00。值得庆幸的是，开发商在英国那边研发中心已经安装了相同的环境，并且也模拟出了我们这边的故障情况。英国研发中心给这边的反馈信息是，他们肯定能找到问题的原因，但是需要时间，最多再给他们两天时间就足够了。而目前对于这个项目来说，最缺乏的也是时间。

我到达现场时，会议室里坐得满满的，运营商代表、Sun公司的技术人员、开发商的测试小组和集成商的技术和商务人员都在等着最后的结果。今天晚上是最后期限，如果无法解决这个问

题，可能会导致项目失败。今天在场的所有公司都可能面临不可承受的失败，所以会议室里的气氛很紧张。运营商的项目经理姓侯，看见我坐下，马上坐到我的身边，他说："我们为这件事已经折腾了 4 天了，你是我们最后一棵救命稻草，千万不要让我失望啊。"

在车上我就一直在想如何处理这个问题。我让同事帮我查了 Oracle 的 bug 数据库，希望能够找到 Oracle 10g 在 Sun 10 上关于性能的 bug。在我走进大都会广场时，深圳那边的反馈结果是没有发现明显的和 Solaris 10 相关的 bug。刚才在路上，我理了理思路。这种情况，应用和数据库、操作系统混合在一起，要在一晚上找到并解决问题也确实有些难度，因此在晚上的分析过程中，不能走太多弯路，否则时间就不够了。

通过大家的介绍，我对这个系统的第一印象是，数据库性能可能存在问题，这个性能问题很可能和 Solaris 10 有关。而从同事给我的信息里，在 Solaris 10 上，Oracle 10.2 没有明显的和性能相关的 bug。因此，按照我以前处理类似问题的惯例，我决定还是从应用入手分析问题。

从应用入手的想法一到客户现场就被打乱了，到达现场后，我首先简单了解了问题以及以前他们做的一些工作。这个问题是几天前发现的，开始定位为 Oracle 的问题，这段时间里已经有好几批 DBA 对数据库进行了各种各样的优化，但是没有任何效果。他们也测试了写一个小的程序，分别在小型机上和微机上跑，测试结果是微机快于小型机。具体之前做了哪些优化，由于前面的同行没有留下任何文档，无从考证，所以我只能从头开始。

在会议桌上找了一台空着的机器，我首先检查了 AWR 报告，检查了操作系统的情况（在这里我犯了一个明显的错误，由于网管对外来电脑设置了一些障碍，使用自己的电脑需要一些额外配置，我没有坚持用自己的电脑，而是用了一台客户的电脑，这个疏忽后来证明是十分错误的，我也因此耽误了很多时间），发现操作系统的物理 I/O 很高，并且从 AWR 报告上发现重做日志相关的性能存在问题，但是物理文件的 I/O 基本正常。很快我确定了第一步的调整思路，对重做日志相关的配置和参数进行了调整，加大了重做日志文件的大小，调整了 Log Buffer 的大小。不过调整后，数据库的性能没有明显的改善，应用的问题依旧。

这个时候,现场开发商的 DBA 向我提供了一系列文档,大多数都是 Solaris 10 上跑 Oracle 10g 的一些问题和 bug 的资料。有 Sun 的官方文档，也有网上搜到的网友的博客和帖子。我简单浏览了一下，初步判断这些资料和现场的问题没有直接的关联。虽然我第一时间否定了这些资料，但是客户的思路也对我有了一定的影响，使我从应用入手的思路有了一定的改变，更加侧重于 Oracle 10g 和 Solaris 配合的问题。使我改变思路的主要有以下几点。

(1) 就是这个系统使用了套装软件，现场人员连应用访问了哪些表，跑了哪些 SQL 都不清楚。我从 AWR 报告中看到的 Top SQL 的情况来看，SQL 没有明显的问题。如果我还是从应用入手进行分析的话，缺乏必要的支持。

(2) 从初步的分析来看，数据库存在性能问题。特别是 log file parallel write 的等待高达 1 460 ms。

(3) 操作系统的 I/O 繁忙程度达到 95%。

(4) 现场人员经过 4 天的分析，所有的线索都指向 Solaris 10。

(5) 应用通过调用 PL/SQL 的接口访问数据库，而接口全部是 WRAP 过的代码，很难针对代

码进行分析。

这个时候，已经是晚上 9 点多了，目前的优化重点是和 I/O 相关的性能问题。匆忙吃过快餐，对目前系统的情况有了一定了解后，开始进行一系列测试。为了采集更多的信息，我安装了一个 STATSPACK。经过测试发现，重做日志切换的时间十分长，可以高达 15s（重做日志文件的大小为 2GB）。从 STATSPACK 报告上看，I/O 还是存在一定的问题。日志文件同步和日志文件并行写的等待还是十分严重。在此阶段，设计了一系列测试程序测试系统的性能。通过测试确认了系统的 I/O 性能确实存在一定的问题，其中重做日志相关 I/O 十分糟糕。我和 Sun 的工程师讨论了一下，他的结论是，存储的条带化处理比较简单，整个存储对于频繁而小的 I/O 处理性能不佳。他的分析和我看到的重做日志写入性能不佳是吻合的，大数据量的写入，比如用 dd 命令来测试硬盘读写速度并不慢。

此时已经是晚上 11：00 了，而现在还丝毫看不到找到问题的希望，现场所有的人都极为紧张，悲观的情绪也弥漫开来。在这种压力下，我已经完全偏离了预先设定的方向，决定把解决 Oracle 和操作系统的配合放在第一位。我亲自进行了测试，发现插入数据的速度并不慢，而提交所消耗的时间很长，这是十分不正常的。通过 10046 分析，看出主要的等待对象是 log file sync。由于 Sun 的负责存储的工程师不在，所以无法得到更进一步的关于存储配置的信息。

时间已经接近 12：00 了，对于 I/O 的分析也取得了一定的进展。在启用 Direct I/O 后，I/O 的性能得到了较大的提升，log file sync 等待时间也从 1000 多毫秒下降为 100～200 毫秒。虽然与正常情况相比，log file sync 仍然偏高，但是已经没有提交需要几秒钟的现象了。从简单的插入测试来看，性能已经有了极大的提升。我要求客户在小型机和微机上分别测试一个大批量插入数据的小程序，发现小型机的性能已经明显高于微机的性能。这个测试结果一出来，所有的人都为之一振。但是接下来的测试结果令人沮丧，应用的测试结果依旧，性能没有任何提升，仍然是 48 秒，一分半，……

在这种情况下，几乎所有人都感到失去了信心。一直坐在我身边的侯经理拿出手机，开始联系其他的外援。我能够很清楚地听到他在和一个号称能够协助解决这个问题的 Sun 方面的高手在谈价钱，那个人开价是解决问题 4 万块钱，没解决收 4 000 块车马费。客户和开发商协商了一下，大家都觉得开价太高，而且那个人并不承诺一定解决问题。听了那个哥们的漫天要价，我突然觉得自己的要价是不是太低了，刚才 Andy 和我说是只有一晚上的时间，如果搞定，客户愿意出 6000 块钱。

这个时候已经快 1：00 了，侯经理拍拍我的肩膀说："老兄，你再继续努力，大家都指望你了，我明天上午有个会，所以必须回去睡一觉了。你如果有好消息，一定给我发个短信。"

虽然问题还是没有解决，不过看着这个测试结果，我却有种突然惊醒的感觉。从目前的测试情况来看，数据库的整体性能至少提升了 30%以上，而应用没有任何提升。好像应用的性能和数据库的性能无关似的。这个现象说明我们以前的主攻方向可能是完全错误的，影响应用性能的关键不在数据库方面，也不在操作系统方面，而是在应用本身。此时英国方面已经准备下班了，开发商的代表问我有没有可能在一个小时内搞定，否则英国那边配合我们会有问题。我说无论如何英国那边一定要有一个人配合我们，希望他们最好能够加班，因为我有个感觉，问题应该是出在

应用上，而且我有信心在今天晚上找到它。经过多方面的协商，英国研发中心那边有个哥们终于答应回家后通过 VPN 继续为我们提供支持。

这时我才想起来使用我的电脑，这样我可以利用自己的一些工具对系统进行分析。还好 Sun 的工程师还在现场，我马上让他帮我设好了上网的一切权限。我决定使用 profiler 工具对 PL/SQL 代码进行跟踪。虽然代码是 WRAP 过的，但是我们起码能够看到函数名以及代码的行号。我需要开发商协助我写一个小的测试程序，模拟创建节点的操作。然后用 profiler 对这个测试程序进行跟踪。在测试进行之前，我首先在测试环境中安装好 profiler 工具。实际上，profiler 工具所需的安装脚本本身在 rdbms/admin 目录下就有，只需要在 sys 账号下执行一下 profload.sql，然后在要测试的账号下面跑一下 proftab.sql 就可以了。

在开发商的协助下，我们很快用 PL/SQL 写出了一个模拟调用 API 的小程序，这个小程序将作为我们测试的依据。首先，我编写了一个 PL/SQL 小程序，调用这个接口：

```
declare
    err number;
    errcode number;
    iotest varchar2(100);
begin
 for i in 1..20 loop
   pkgsession.configuresession();
test(iv_name=>'m3m'||to_char(i),in_execount=>1000,on_errorcode=>err,ov_errortext=>
iotest);
   insert into temp1 values (systimestamp);
   commit;
  end loop;
end;
/
```

通过这个小程序，我们发现，第一个循环 6s 就结束了，第二次循环消耗 10s，今后每次循环都以四五秒钟的时间递增。而如果我们重新开一个会话，第一次循环又变为 6s。如此规律在每次测试中都重现，看样子我快要抓住问题的关键了。此时已经是凌晨 1：00，英国那边也传来一个临时措施，就是每次调用该接口后都重新连接一下数据库。从目前的情况来看，这个可以作为备选方案，但是这并不是解决问题的正确途径，因为连接数据库也是一个开销十分大的操作，肯定会影响最终的测试结果。

从上述测试更加确定了应用存在问题，下一步就是如何找到到底问题出在哪里。这时 profiler 就可以派上用场了。通过 profiler 编写如下脚本：

```
declare
    err number;
    errcode number;
    iotest varchar2(100);
begin
 for i in 1..3 loop
   err:=DBMS_PROFILER.START_PROFILER ('test'||to_char(i));
   pkgsession.configuresession();
```

```
test(iv_name=>'Aba'||to_char(i),in_execount=>1000,on_errorcode=>err,ov_errortext=>
iotest);
   err:=DBMS_PROFILER.STOP_PROFILER ;
   insert into temp1 values (systimestamp);
   commit;
  end loop;
end;
/
```

从 profiler 的测试结果，我们发现第一个循环和第二个循环调用的顺序（无法看到源码，只能看到包的名称、存储过程的名称以及每行代码的行号，通过行号分析）是不同的，因为程序执行的行数和行号完全不同，而从第二次开始每次调用执行的程序从行号上看，所有调用顺序是完全相同的。

我们将第二次和第三次执行的所有行的统计信息复制到一个 Excel 表中，然后按照每行的执行时间排序。经过排序后，我们发现，第二次和第三次执行的统计信息按照执行时间排序后，排在前三位的行的行号是完全相同的，不过这三行中第三次执行的次数比第二次执行次数多了 100 万次，这 100 万次消耗的时间正好是 5s，正是第三次执行比第二次执行慢的时间。

这可能是问题所在了。分析结果一出来，实验室中所有的人都兴奋起来。在通知英国方面的同时，我们马上在笔记本电脑上进行了相同的测试。测试结果发现，在笔记本电脑上，根本找不到调用这个包的代码。笔记本电脑上根本就没有这些操作，这就是说笔记本电脑和小型机上跑的应用是有区别的。经过确认，两个环境上安装的是完全相同的软件。以我多年软件开发的经验，虽然两个环境安装的是相同的软件，但是可能存在某种开关，在笔记本电脑上关闭了引起小型机上性能下降的调用操作。看样子这回找到了问题的根源，这时已经是凌晨 3 点多了。我让和英国沟通的哥们通过 MSN 告诉英国方面，他们的重点应该放在开关上，因为配置文件已经确认过很多次了，肯定不会是配置文件存在不一致，肯定是软件内部的一个开关引起的。这个开关可能在某些平台上的默认值不同。

半个小时后，英国方面的研究结果反馈回来了：这三行代码是测试使用的，本身在生产环境中就不应该执行。他们建议我们在安装这个软件的 Oracle 用户上关闭一个策略。我们按照英国方面的指引，关闭了那个策略，然后马上启动了那个测试程序。大家都盯着那个测试工程师的脸，希望从他的脸上看到笑容。突然那个哥们举起拳头大叫了一声“搞定”，房间里爆发出热烈的掌声，大家都激动地涌到测试终端的面前。

现场工程师马上通过 MSN 把这边的测试结果告诉了英国那边，英国那边也是兴奋异常。由于这次测试的结果可能决定了这个产品在中国的生存问题，所以英国研发中心虽然下了班，还是有几个人放弃了休息，自发留在办公室里协助我们，他们得知问题得到解决的消息，也十分兴奋，决定等会就一起去酒吧喝几杯。问题终于得以解决。虽然大家忙了一晚上，但是问题得到了圆满的解决，大家的心情还是比较愉快的。

现在已经是凌晨 4：00 了。从刚才测试的结果来看，平均创建一个节点的时间是 5s，为了完成这次测试，需要预先装载 20 万个这样的节点。以目前的速度来看，还需要几百个小时才能够

完成数据的装载操作。气氛一下子又跌倒了谷底。我想了想，我建议他们不要使用程序装载数据，而是直接写一个 PL/SQL 存储过程，批量载入数据，加上 bulk 操作，估计可以在 3～5 个小时完成数据装载。听到这个消息，大家又兴奋起来了，不过问题又来了，英国那边的人已经一个也联系不上了，而广州这边没有一个人知道装载节点需要对哪些表做些什么操作。

我说这个倒不怕，通过设置一个 10046 Trace，就可以找出生成一个节点需要执行哪些 DML 操作了。通过 10046，我们很快把所有的 DML 操作都找了出来，而且他们的开发人员对 PL/SQL 中使用 bulk 操作很熟悉，那后面就很好办了。我和他们交代了一下，就离开了。昨天订了一个酒店，也基本没用上。我开房间时，服务员告诉我，现在广州开交易会，你预定的是昨天的房间，所以虽然你 6 点以后才入住的，但是我们还是要按照昨晚入住算房费。我说这都无所谓了，反正有人买单，你给我找个安静点的地方就行了。

一看到舒服的床，就什么都不想了，我一沾枕头就睡着了，一觉睡到中午 12：30。做了这么一个激动人心的项目，居然能够沾枕头就睡，看样子服务做长了，对优化已经没有什么新鲜感了。

优化小技巧　本案例的启示

这是一个十分典型的性能优化案例，也是一个给人很多启发的案例，从中我总结了以下几点。

(1) 处理类似的案例，一定要有很好的预案，不能因为一些未预料到的细节而轻易改变方案。在这个过程中，最初我设定的预案没有问题，以应用为核心进行分析。我也一度偏离了正确的方向。如果一直坚持以应用为核心的分析原则，可能可以提前几个小时解决这个问题。

(2) 作为 DBA，应该了解小型机、硬件、操作系统、数据库、应用开发。对于这种问题，一个不了解应用开发的 DBA 是很难找到根源的。

(3) 要敢于坚持自己的正确观点，要有足够的能力判断自己是在做正确的事情，不要轻易被其他声音淹没。其他技术人员的观点可以参考和借鉴，但是应该能够对这些信息进行分析，而不能不加思考地全盘接收。

(4) 在这个处理过程中，客户也给我提醒过，做调优，能提升 20%就不错了，而现在的问题已经不是 20%那么简单。而我开始时过多地把时间放在了调优上面，后来发现 log file sync 虽然很严重，但是也仅仅占到 10%左右。而解决这个问题花掉了超过 60%的时间。

(5) 要活用 Oracle 的工具。profiler 一般是用来调试 PL/SQL 程序的，但是这时，变成了问题查找的利器。

(6) 事后大家在讨论,为什么这个应用的问题，英国那边有相同的环境，能够重演相同的问题，又有那么多开发人员介入，没有找到问题的根源，而我们能够在这么快的时间就解决了问题？最后的结论是，英国那边缺乏有经验的 DBA。很多事情，DBA 和开发人员一起才是最好的组合，仅仅有一样是不够的。这个案例是最好的证明。

注　本节案例使用的一个 Oracle 工具 profiler，在本书 5 月 20 日的“优化小技巧”中有详细介绍，请参考。

后记，也算结束语

到今天，这本书也就结束了，其中介绍了一个实际的优化案例。有些读者可能会感到里面的内容有些乱，也有些人可能感觉抓不到头脑，不知道我想表达些什么。确实，这本书如果当做小说来看，未免太枯燥；如果当做纪实文学来看，又不能弘扬主旋律；作为技术书来看，里面又缺乏有分量的技术。实际上，我在刚开始就说过，本书不是一本介绍 Oracle 技术的书，因此读者并不需要像读技术书那样一字一句地去抠一些字眼。通过这本书，老白给大家展现了一个优化项目，并且教给大家一些系统优化的方法和技巧，特别是在优化项目中如何控制项目，如何让整个优化工作按照自己的预想进行。

粗粗看一遍本书，可能会感觉一个优化项目里好像也没有什么，实际上如果你能够仔细回味一下，或者把本书仔细翻阅两遍，你会发现在平淡的工作中还隐藏了很多内容。实际上，优化项目里面并没有十分高深的技术手段。本书中，老白和老于他们采取的所有技术手段都是大家以往学习过的，都是一些耳熟能详的东西。没有做过优化项目的人，可能觉得做一个优化项目无从入手。也有一些人可能感觉优化项目里使用的技术不过如此，优化实际上也不是什么很难的事情，不过如果你真的去做一个优化项目，就会感觉好多地方都无法把握。确实是这样，优化项目中最难的是准确的判断，准确的把握。在前期的数据采集和分析阶段，你真的已经了解了系统的现状和主要问题吗？实际上，要回答这个问题并不简单，这需要很丰富的经验。如果你对数据库的原理融会贯通了，那么很容易从数据库运行的原理去考虑问题，也就很容易找到目前系统存在什么问题。否则，你可能无法抓住主要问题，真要是这样，这次优化基本上就报销了。

其实本书各部分的内容都是我精心安排的，哪怕是一些好像没有什么技术内容的章节。比如“电脑坏了”那一节讲的虽然是我的电脑坏掉了，如何去修理，如何通过小齐从 HP 沈阳办事处借了一台电脑。实际上，这一节的安装软件那方面的内容，介绍了一个 DBA 的电脑里最常用的工具软件。我虽然不是一个唯工具论者，但是我认为工欲善其事，必先利其器。适当地使用工具，合理地使用工具是十分重要的。一个只会使用 SQL*Plus 的 DBA 并不一定就是一名优秀的 DBA。

“世博园一日游”那一节实际上是要向读者说明 DBA 和客户沟通时应该掌握一定的技巧，做 DBA 不能光有技术，而且需要通过一些小手段，让客户对你产生信任，以及出现危机时，如何去化解和扭转。

对于每个 DBA 或者想成为 DBA 的读者来说，DBA 日记从另外一个侧面介绍了一名 DBA 应该如何去工作，在 DBA 工作和生活中有哪些需要注意的地方。这也是我写作本书的初衷，因为

我觉得目前市场上缺乏这样一本书。我希望大家可以愉快地阅读本书，也希望读者能在轻松的阅读中获得一些知识，掌握一些技巧，仅此而已。我更希望有更多的 DBA 把自己真实的经历和感受写出来，展现给大家，希望 DBA 的圈子能够更加开放。

Oracle优化日记

一个金牌DBA的故事

"我知道很多DBA都有自己收藏的一本秘籍，包括处理各种Oracle问题的必杀术，其中关于性能调优的特别多。可惜这里面有很多道听途说的成分，或者知其然而不知其所以然。反观此书，有很多活学活用的例子；对于各种技巧的适用范围，也尽量阐述清楚。从最后成书的内容看，作者还是比较负责的。书中的指令，大都经过上机验证。"

——Ben Wang，澳大利亚知名Oracle技术高手

数据库优化一直是DBA感到十分高深的问题，很多DBA学习了大量优化技术，参加了大量培训，但是仍然感觉在实际工作中无从入手。实际上，在数据库优化工作中，比技术更为重要的是方法。

本书以作者亲身经历的一个大型优化项目为原型，用日记的形式讲述了项目实施的全过程。生动曲折的项目故事，扣人心弦的尖峰时刻，一一再现了几个经验丰富的老DBA分析数据库性能问题时的真实场景。书中不仅剖析了优化过程，还展现了在优化项目中处理纷繁的关系的技巧和方法。伴随精彩的故事情节，作者还介绍了DBA常用的优化技巧，使读者在大呼过瘾的同时，亦能分享到金牌DBA的宝贵经验。

恩墨科技创始人 Oracle ACE Director 盖国强倾力推荐

图灵网站：www.turingbook.com　热线：(010)51095186
反馈/投稿/推荐信箱：contact@turingbook.com
有奖勘误：debug@turingbook.com

分类建议　计算机/数据库/Oracle

人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-23071-3

ISBN 978-7-115-23071-3
定价：55.00元

看完了

如果您对本书内容有疑问，可发邮件至contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这里可以找到我们：

微博 @图灵教育 : 好书、活动每日播报
微博 @图灵社区 : 电子书和好文章的消息
微博 @图灵新知 : 图灵教育的科普小组
微信 图灵访谈 : ituring_interview，讲述码农精彩人生
微信 图灵教育 : turingbooks